Improving Application Performance in the Emerging Hyper-converged Infrastructure

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Hao Wen

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

David H.C. Du

April, 2019

Acknowledgements

I would like to express my sincere appreciation to my advisor, Prof. David H.C. Du. He taught me the capabilities of critical thinking and carrying out research independently. I learned from him how to identify valuable research issues and attack them directly and elegantly. His mentorship leads my way to a deep understanding of the computer science field, invokes my tremendous interest in doing research, and will have long lasting impact on my future career.

I would also like to thank Professor Rui Kuang, Professor Abhishek Chandra, and Professor Soheil Mohajer for serving as my committee members and for their invaluable comments and suggestions.

It is an honor for me to be a member of the Center for Research in Intelligent Storage (CRIS). I would like to thank my CRIS mentor, Doug Voigt (retired, previously with HP Enterprise), and Ayman Abouelwafa (currently with HP Enterprise) for their great efforts in collaborating with me to conquer research problems. I also want to thank CRIS members: Zhichao Cao, Fenggang Wu, Baoquan Zhang, Bingzhe Li, Ming-Hong Yang, Chai-Wen Hsieh, Jim Diehl, Hebatalla Eldakiky, Jinfeng Yang, Yaobin Qin, Ziqi Fan, Xiang Cao, Xiongzi Ge, Alireza Haghdoost, Weiping He, and Manas Minglani. I learned so much through the discussion and collaboration. Thanks for their help and support.

Finally, I would like to thank NSF I/UCRC Center for Research in Intelligent Storage and the following NSF awards 1439662, 1525617, 1536447, 1708886, 1763008, and 1812537 for supporting my research.

Dedication

To my family.

Abstract

In today's world, the hyper-converged infrastructure is emerging as a new type of infrastructure. In the hyper-converged infrastructure, service providers deploy compute, network and storage services on inexpensive hardware rather than expensive proprietary hardware. It allows the service providers to customize the services they can provide by deploying applications in Virtual Machines (VMs) or containers. They can have controls on all resources including compute, network and storage. In this hyper-converged infrastructure, improving the application performance is an important issue. Throughout my Ph.D. research, I have been studying how to improve the performance of applications in the emerging hyper-converged infrastructure. I have been focusing on improving the performance of applications in VMs and in containers when accessing data, and how to improve the performance of applications in the networked storage environment.

In the hyper-converged infrastructure, administrators can provide desktop services by deploying Virtual Desktop Infrastructure application (VDI) based on VMs. We first investigate how to identify storage requirements and determine how to meet such requirements with minimal storage resources for VDI application. We create a model to describe the behavior of VDI, and collect real VDI traces to populate this model. The model allows us to identify the storage requirements of VDI and determine the potential bottlenecks in storage. Based on this information, we can tell what capacity and minimum capability a storage system needs in order to support and satisfy a given VDI configuration. We show that our model can describe more fine-grained storage requirements of VDI compared with the rules of thumb which are currently used in industry.

In the hyper-converged infrastructure, more and more applications are running in containers. We design and implement a system, called k8sES (k8s Enhanced Storage), that efficiently supports applications with various storage SLOs (Service Level Objectives) along with all other requirements deployed in the Kubernetes environment which is based on containers. Kubernetes (k8s) is a system for managing containerized applications across multiple hosts. The current storage support for containerized applications in k8s is limited. To satisfy users' SLOs, k8s administrators must manually

configure storage in advance, and users must know the configurations and capabilities of different types of the provided storage. In k8sES, storage resources are dynamically allocated based on users' requirements. Given users' SLOs, k8sES will select the correct node and storage that can meet their requirements when scheduling applications. The storage allocation mechanism in k8sES also improves the storage utilization efficiency. In addition, we provide a tool to monitor the I/O activities of both applications and storage devices in Kubernetes.

With the capabilities of controlling client, network and storage with hyper-convergence, we study how to coordinate different components along the I/O path to ensure latency SLOs for applications in the networked storage environment. We propose and implement JoiNS, a system trying to ensure latency SLOs for applications that access data on remote networked storage. JoiNS carefully considers all the components along the I/O path and controls them in a coordinated fashion. JoiNS has both global network and storage visibility with a logically centralized controller which keeps monitoring the status of each involved component. JoiNS coordinates these components and adjusts the priority of I/Os in each component based on the latency SLO, network and storage status, time estimation, and characteristics of each I/O request.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Cloud computing platforms are widely adopted by enterprise, serving as the IT infrastructure today. With cloud computing, computer system resources are delivered on demand to users, without direct management by the users. In today's cloud infrastructure, virtualization technology is the building block. When users deploy applications in cloud, the cloud provider will allocate essential compute, network and storage resources to support the running of applications. User applications will be running in Virtual Machines (VMs) [1, 2] or containers [3, 4]. At meantime, virtualized storage services are providing either block storage (e.g., Amazon EBS [5]) or object storage (e.g., Amazon S3 [6]) to the applications. The virtualized network is connecting from application to application, and connecting between applications and storage.

Despite of the success of cloud, there are some issues in today's cloud infrastructure. Today's cloud is built upon proprietary expensive servers, switches and storage systems. These proprietary hardware only provide limited capabilities of customizing the functions that they can perform. These hardware usually come from different companies and have different interfaces and management systems. It further increases the difficulties of controlling these hardware. In the cloud infrastructure, administrators do not have full control within the cloud. They do not have control outside of the cloud either. Recently, as Internet of Things (IoT) is rapidly developing, edge devices such as speakers, light switches, cell phones, smart watches, etc. start to involve in various compute jobs [7, 8, 9]. Some people are trying to run a compute job like DNN (Deep Neural Network) inference in both edge devices and cloud to achieve a better performance. They can

choose to run a DNN job completely in cloud [10], completely at edge [11], or in both edge and cloud [12, 13]. Some research are focusing on customizing the edge devices to accelerate the machine learning process [14, 15] or achieve the low-cost design for machine learning [16, 17, 18, 19, 20, 21, 22]. Therefore, it becomes important for the administrators to extend their control outside of the cloud. Recently, a new type of infrastructure called hyper-converged infrastructure is emerging. Service providers tend to deploy compute, network and storage services on inexpensive hardware rather than expensive proprietary servers, switches or storage systems. These inexpensive hardware only provide very basic functions and work as white boxes, allowing service providers to customize the services they can provide by deploying applications in VMs or containers. They can have controls on all resources including compute, network and storage at various components of the infrastructure.

The hyper-convergence has opened up new research opportunities. For example, people are trying to decouple network functions like firewall or encryption from dedicated switches and routers, and move the functions to virtual servers. This collapsing multiple functions into a single physical server reduces the cost. If a customer wants a new network function, the service provider can simply launch a new VM or container to perform that function. This type of technology is called network function virtualization (NFV) [23]. Typical network functions include intrusion-detection system, firewall, in-network congestion control [24], in-network key value store, multi-path TCP detection [25], network consensus [26, 27], etc. Inspired by NFV, we can study whether we can virtualize storage management functions with the customization capabilities provided by hyper-convergence. Possible candidates can be disk management [28, 29], storage reliability [30, 31], I/O scheduling, caching [32], privacy protection [33], data backup, snapshot, data encryption, data deduplication [34, 35, 36], data analytics, etc. For disk management, we can design a more flexible disk management system so that it can control and manage heterogeneous types of storage disks, including the traditional disks like HDD, SSD, emerging key-value Ethernet drives like Kinetic drive [37], and the emgerging high capacity disks like SMR/IMR disks [38, 39, 40]. In addition, data analytics functions can be good candidates of virtualized storage functions. Modern storage systems have data analytics capabilities to intelligently configure various storage functions. They may apply machine learning [41, 42, 43, 44, 45, 46, 47, 48] and

data mining [49, 50, 51, 52, 53, 54, 55] technology to improve the data allocation efficiency, data backup performance, etc. Moreover, performance evaluation [56, 57] on such hyper-converged infrastructure is also an important research topic to investigate different applications in this infrastructure.

In this hyper-converged infrastructure, improving the performance of applications is a very important issue. When running applications in this infrastructure, users usually have service-level objectives (SLOs) on performance as part of their service-level agreement (SLA) requirements. It is important for service providers to allocate appropriate resources and provide essential services to support the running of applications.

A lot of issues need exploration in improving applicaiton performance in the hyper-converged infrastructure. In this infrastructure, applications can run in VMs. How we can improve the performance of applications in VMs when accessing data is a valuable problem. Recently, container becomes a popular virtualization technology. More applications are running in containers. Then how we can improve the performance of applications in containers when accessing data is also an important issue. As we have control in both network and storage with hyper-convergence, we may think about how to provide a systematic way to control client, network and storage in order to improve data access performance when applications access storage through network. In addition, with NFV in the hyper-converged infrastructure, we can study the impact of NFV on the performance of applications. Once we have virtualized the storage functions, we can also study the impact of storage function virtualization on the performance of applications. After we have the capabilities of customizing network functions and storage functions by creating VMs or containers, we can investigate how to allocate resources to user' applications, network functions and storage functions in a data center to optimize the overall performance or cost. In this thesis, we focus on three topics: (1) Improving the performance of applications in VMs when accessing data; (2) Improving the performance of applications in containers when accessing data; (3) Improving the performance of applications when accessing storage through network. Figure 1.1 highlights these three topics in the emerging hyper-converged infrastructure. Users run their applications in VMs or containers. Applications may reside on a single host or across multiple hosts. Applications access data through data center network or Internet.

In the hyper-converged infrastructure, administrators can provide desktop services

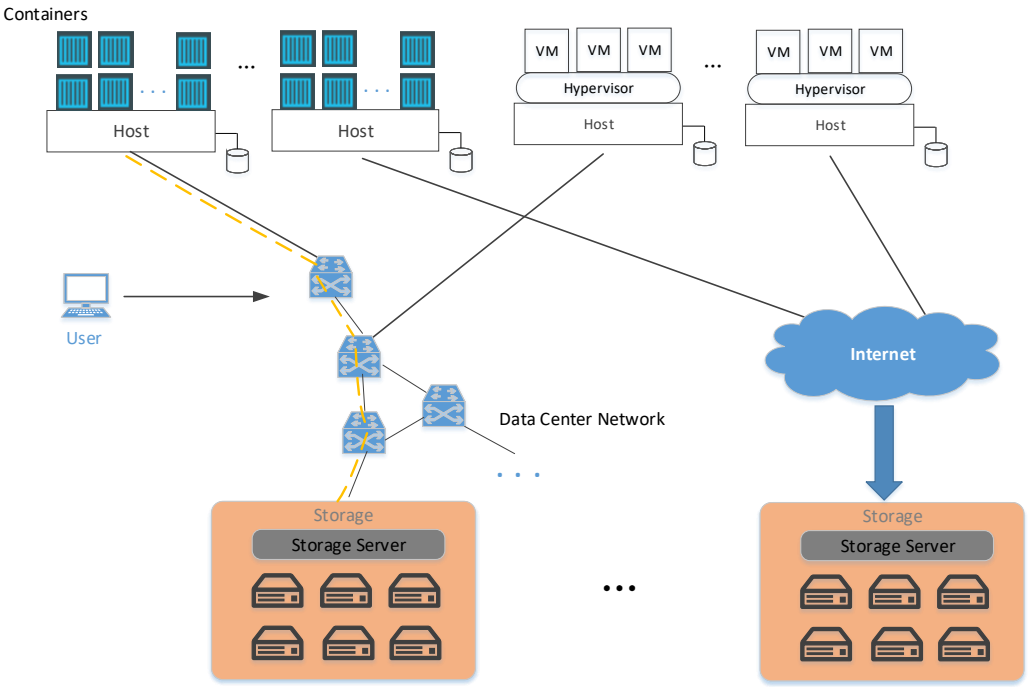


Figure 1.1: Cloud Storage Environment

by deploying Virtual Desktop Infrastructure application (VDI) based on VMs. In the first topic, we investigate how to identify storage requirements and meet the requirements with minimal storage resources for VDI application [58, 59]. VDI is based on the typical VM environment. In the VM environment, a hypervisor is installed on a physical server to perform the role of virtualizing the underlying hardware resources and managing the VMs running on the hypervisor. In each VM, there is an isolated operating system hosting the runtime environment for applications inside the VM. Multiple VMs on the same physical host share resources. VDI is a typical and also prevalent VM application. VDI manages desktops in a data center, and presents desktops to remote users like running desktops locally. Users of VDI can access virtual desktops and their data from anywhere at anytime by simply logging into the virtual desktops through a thin client.

In the hyper-converged infrastructure, more and more applications are running in containers. In the second topic, we study how to support applications with various storage service level objectives (SLOs) in the Kubernetes environment which is based on

Docker containers. Linux containers [3, 4] is one most popular containerization technology. It is a virtualization method for running multiple applications in isolated systems (i.e., containers) on a host using a single Linux kernel. Linux containers were initially released in 2008, but their use soared after Docker's release in 2013 [60, 61]. Docker is a platform for creating, deploying, and running applications in Linux containers. With Docker containers, an instance or a function module of an application is able to run as an individual micro-service in containers. Different from VMs, which require each VM running an individual operating system, containers on the same host will share the same operating system kernel. Each container can have its own libraries and binaries. In order to deploy and manage applications in Docker containers across multiple hosts, people use a container orchestrator to perform cluster management and orchestration (i.e., selecting which cluster nodes will host containers). Kubernetes (k8s) is one prevalent container orchestrator. K8s offers automatic deployment, maintenance, scaling, and resource management for applications. In this new environment with containers and k8s, the current storage support for containerized applications is not well studied. With a clear understanding of the current storage support in the container environment, we can then study how to improve the data access performance for containerized applications. In this topic, we first study the current storage support in k8s and discuss its limitations. We then propose a system that can efficiently support applications with various storage SLOs along with all other requirements deployed in the Kubernetes environment based on Docker containers.

In the third topic, we study how to coordinate different components along the I/O path to ensure latency SLOs for applications in the networked storage environment [62]. In recent years, more and more data of applications are stored at remote storage. This kind of storage connecting to client through network is also called networked storage. Cloud storage [6, 63, 64, 65], data center SAN, NAS, and object storage are typical networked storage. With the involvement of network in data access, the latency sensitive I/O access will be affected by the network. When accessing storage, an I/O request will go through the client side I/O stacks, transit through the network, traverse the storage servers and finally be served by storage devices like disks. The response of the request also has to go through these components on the reverse path back to the client. This

long I/O path and the diverse components along the path result in increased end-to-end management complexity. Different from the traditional methods of excelling control on a single component, the hyper-convergence infrastructure gives us potentials to control client, network and storage together. We believe considering all the involved components, and controlling client, network and storage in a systematic way are essential to achieve a better data access performance. In this topic, we discover that an isolated control on either network or storage is not efficient in ensuring data access performance. Thus, we propose a system that coordinates different components along the I/O path to ensure latency SLOs for applications that access data in remote networked storage.

The rest of this work is organized as follows. Chapter 2 describes that how we identify storage requirements and meet the requirements of VDI applications. Chapter 3 discusses how we enhance the storage of Kubernetes to support applications' storage SLOs in containers. Chapter 4 discusses how we improve the I/O latency of applications in networked storage environment. Chapter 5 concludes this thesis.

Chapter 2

Meeting Storage Requirements of VDI Applications

2.1 Introduction

With the rapid development of virtualization technology, traditional data centers are increasingly replacing dedicated physical machines with virtual machines (VMs) to provide services. Apart from improving hardware utilization, virtualization enables seamless migration of applications to a different physical host for the purpose of load balancing, planned software/hardware upgrades, etc. To avoid migrating data along with the in-memory state of the virtual machines, virtual machine data is typically stored on shared storage. In the shared storage architecture, multiple VMs/applications compete with each other for input/output (I/O)resources and capacity of the storage system. To reduce costs, data center administrators need to determine how to meet the storage requirements of these VMs with the minimum amount of required resources.

In this chapter, we investigate how to identify storage requirements to support one popular type of VM application, Virtual Desktop Infrastructure (VDI) [2, 66, 67, 68, 58]. VDI runs multiple VMs with different operating systems and applications on several physical servers in a data center. This use of VMs is also known as desktop virtualization with each instance called a virtual desktop. Current VDI sizing work [69, 70, 71, 72] is unable to give an accurate description of the storage requirements of virtual desktops. They either use rules of thumb to guide storage provisioning [69] or test the

performance of their storage array under a given fixed number of VDI instances [72]. To ensure VDI users to not see degraded performance in practice, administrators typically over-provision storage resources which may cause some waste. In addition, how CPU, memory, and storage resources of virtual desktops are configured may have a considerable impact on the I/O behavior of VDI. For example, each virtual desktop may access multiple heterogeneous data disks at different times causing each data disk to see significantly different I/O workloads. Therefore, how physical storage is configured and where these data disks are placed will impact whether the storage requirements are met. Unfortunately, current VDI sizing work fails to give a clear description of VDI configuration and its required storage resources.

When considering VDI performance, CPU, memory and storage can all be potential bottlenecks. We assume enough CPU and memory are provided in a data center. Besides, VMs can be migrated to another host [73] if the current host utilization is high. In this chapter, we focus only on the storage requirements of a given VDI configuration to guarantee good performance.

Our objective is to meet storage requirements of VMs with minimal storage resources. This depends on many practical factors. In this chapter, we mainly focus on the required storage capacity, throughput, and IOPS (I/Os per second). We create a model to describe I/O behavior of a single virtual desktop as well as a group of virtual desktops. With the model, we are able to tell when and where the performance bottlenecks occur. Based on this foundation, we can identify what capability a storage appliance needs in order to satisfy a given VDI configuration and its storage requirements.

To create such a model, we need to know the detailed implementation of VDI, virtual desktop types, and access patterns of virtual desktops. The implementation includes the organization of underlying storage and the composition of each virtual desktop. Considering that there are multiple virtual desktop types, the model should adapt to both homogeneous and heterogeneous combinations of virtual desktops. Each desktop undergoes certain stages (boot, login, active, and logoff) during its life cycle and accesses multiple different data disks at different stages. The access pattern of a virtual desktop is affected by its current stage and the data disks it accesses at different stages. Those data disks have different functions and see distinct I/O access patterns. When large

numbers of virtual desktops arrive at different times, the aggregation effect of I/Os will lead to more variance of storage access patterns.

Our contributions are summarized as follows:

- We describe several different representatives of VDI and discuss their unique storage access patterns.
- We propose a system model to describe the I/O behavior of both homogeneous and heterogeneous configurations of VDI. To the best of our knowledge, this is the first model to do so for real life VDI systems.
- We identify the storage requirements of VDI and determine the bottlenecks on specific target virtual disks at a specific time.
- With the detailed storage requirements, we show how to size a minimum storage configuration that satisfies these VDI requirements.

The organization of this chapter is as follows: Section 2.2 provides a detailed background of VDI and presents its unique characteristics. In Section 2.3, we propose our system model. Section 2.4 analyzes the VDI traces and discusses the storage requirements and bottlenecks generated from the model. Section 2.5 shows the application of our model in real life. In Section 2.6, we discuss future research work on migrating virtual desktops from VMs to Docker containers including some preliminary experimental results showing the feasibility of using Docker containers in VDI. Section 2.7 presents related work and Section 2.8 concludes the chapter.

2.2 Background

Virtual Desktop Infrastructure (VDI) is a virtualization technology to provide desktop environments to remote users. VDI runs desktop operating systems (e.g., Windows, MacOS, etc.) on virtual machines (VMs) in a data center and presents normal desktops to remote users. Thus, a virtual desktop is a desktop environment or desktop operating system running in a VM. A virtual desktop can be associated with multiple different virtual disks. These virtual disks can reside either on local or shared storage in a data center. Virtual desktops are managed centrally in VDI. A user can use client devices

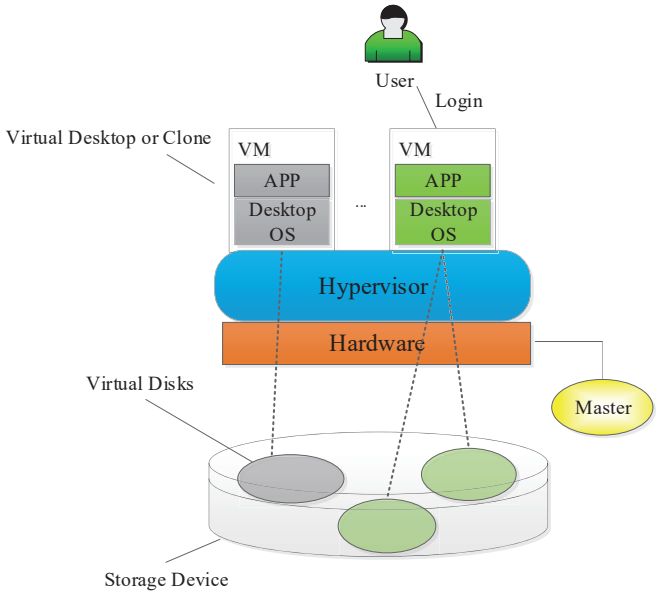


Figure 2.1: An example of VDI architecture.

such as personal computers, laptops, tablets, and mobile phones to connect to and operate his/her virtual desktop. An example of VDI architecture is shown in Figure 2.1. In VDI, all virtual desktops can be thought of as clones. They are clones of a master image. A master image is a VM template from which other virtual desktops originate. Virtual desktops are running in VMs created through hypervisors on physical servers.

In the remainder of this section, we first introduce different clone types. Next, we describe how virtual desktops are assigned to users. We then introduce how we get different virtual desktop types by combining clone types and assignments. Finally, for each virtual desktop type, the associated data disk types are introduced.

2.2.1 Clone Type

There are two main types of virtual desktop clones. One is the full clone. A full clone copies the master image into its own virtual disk (not shared). The other type is the linked clone. Different from full clones, linked clones share the same operating system (OS) data as long as they are linked to the same replica (a clone of the master image). Each replica serves as a common base for a group of linked clones.

2.2.2 Virtual Desktop Assignment

Virtual desktop assignment binds a user with a virtual desktop. There are two main types of virtual desktop assignment: dedicated assignment and floating assignment. Dedicated assignment assigns a virtual desktop exclusively to a certain user. After the assignment, when a user tries to log into his/her virtual desktop, it is always the same VM serving the user. User profiles and user data are permanently saved in its local virtual disks. Both full clones and linked clones can be dedicated. Floating assignment arbitrarily assigns a virtual desktop to a user. Each time a user logs in, he/she may be assigned to a different VM. User profiles and user data are not saved in local virtual disks but are rather saved remotely. Only linked clones can be assigned as floating. Dedicated assignment has the advantages of good virtual desktop launching speed and user data access speed, but floating assignment allows for more flexible resource allocation across the whole system.

2.2.3 Virtual Desktop Types and Their Associated Disks

Combining clone type with assignment gives three kinds of virtual desktops: floating linked clone, dedicated linked clone, and dedicated full clone. Each type of virtual desktop is associated with a different set of virtual disks. According to the usage and types of data stored, there are six types of virtual disks: master image, replica, primary disk, persistent disk, remote repository, and full clone disk. The remainder of this section describes which of these virtual disk types are associated with each of the three virtual desktop types.

Floating linked clone. By definition, each linked clone is linked to a shared replica (a clone of the master image). To provision linked clones, a replica must first be created from the master image. In the linked clone pool, we should provision spare space for multiple replicas with different operating systems. Besides the shared replica, a primary disk contains the essential system data that is needed for each linked clone to remain linked to the shared replica and to function as an individual desktop. Floating linked clones are usually configured not to save user profiles and user data in their local virtual disks. User profiles are preserved in a remote repository independent of the virtual desktop. Each user has his own repository. Typically, they are stored in a NAS

(Network Attached Storage) device. In the remaining of this chapter, we use remote repository and NAS interchangeably.

Dedicated linked clone. Dedicated linked clones include those data disks essential for linked clones: replica and primary disk. However, what is unique about dedicated linked clones is that a separate persistent disk can be configured to store user profiles and user data. This disk is dedicated to a user. Attaching a persistent disk to a linked clone, virtual desktop makes that virtual desktop dedicated to the user. A remote repository is also needed to permanently store user profiles and user data.

Full clone. A full clone is always dedicated. Each full clone is an independent virtual desktop. Therefore, a full clone uses its own full clone disk (its regular virtual disk) to store the operating system, user profiles, and user data.

2.3 System Model

In order to understand the storage requirements of a VDI system, we propose a model to describe a VDI system in a data center. Based on this model, we can infer when and where the storage bottlenecks are. Since virtual desktops run in VMs, we describe the I/O behavior of VMs that are hosting virtual desktops in the model. We first model a single VM and then include different types of VMs to model a large number of VMs in VDI.

2.3.1 VM Life Cycle

A VM in VDI has multiple stages during its life cycle. Each stage shows distinct I/O behavior. Overall, a VM life cycle has four stages: **boot, login, active, and logoff**. In the boot stage, VMs are booting. If many of those virtual desktops are powered on at the same time, or concentrated within a small time period, it becomes a boot storm. For large systems serving virtual desktops across multiple time zones, there can be several boot storms per day. After desktops are powered on, users will log into the desktops. Since the boot stage can be a storm, the login stage can also be a storm just after boot. After users log in, virtual desktops transit to active stage. During this stage, people do their everyday work, e.g., watching videos, reading documents, writing slides, etc. This stage may even be divided into multiple sub-stages depending on the

I/O patterns. For example, in a software development company, people may be busy and are actively generating I/Os between 9am to 5pm during weekdays; from 5pm to 9pm, the number of people working declines; from 9pm to 12am, only a few people are using virtual desktops and their I/O workloads are lighter; finally from 12am to 9am, almost all people have logged off and only sparse I/O activities can be observed; during weekends, the I/O activity is similar to the sub-stage of 9pm to 12am of weekdays. Then, for this company the active stage during weekdays can be further divided into 4 sub-stages: busy (9am to 5pm), average (5pm to 9pm), light (9pm to 12am) and idle (12am to 9am). The active stage during weekends has a single light sub-stage. The reason of creating sub-stages of the active stage is that there may be significant transitions in the workload that require separate descriptions of the workload. Different companies may see different I/O access patterns, as observed in [74, 75]. Logoff is the final stage during a VM life cycle. Users finish their work in VDI and log off from their assigned virtual desktops.

2.3.2 Data Access Sequence

As we discussed in the previous section, different virtual disks are accessed by various types of virtual desktops. Even for the same virtual desktop type, virtual disks may see distinct I/O access patterns at different stages. We will now discuss how each virtual desktop type accesses virtual disks at each stage.

Data accesses of floating linked clones. The storage architecture of floating linked clones is shown in Figure 2.2. Multiple floating linked clones are running on hypervisors across different physical servers. On each server, there may be one or multiple master images to generate linked clones. Each floating linked clone is linked to a shared replica. Additionally, a primary disk is bound to a floating linked clone. These virtual disks are grouped into different data stores. Typically, each data store only has one type of virtual disk. In theory, there are no rules regarding which types of storage (SSD, HDD, etc.) should host those virtual disks. Administrators can opt to place those virtual disks on any type of storage. However, without detailed analysis and characterization of VDI demand, it is hard for the administrator to give a good placement that satisfies VDI storage requirements. This is the focus of our chapter. The green lines and red lines in the figure show the data accesses in the boot and login stage, respectively. When a

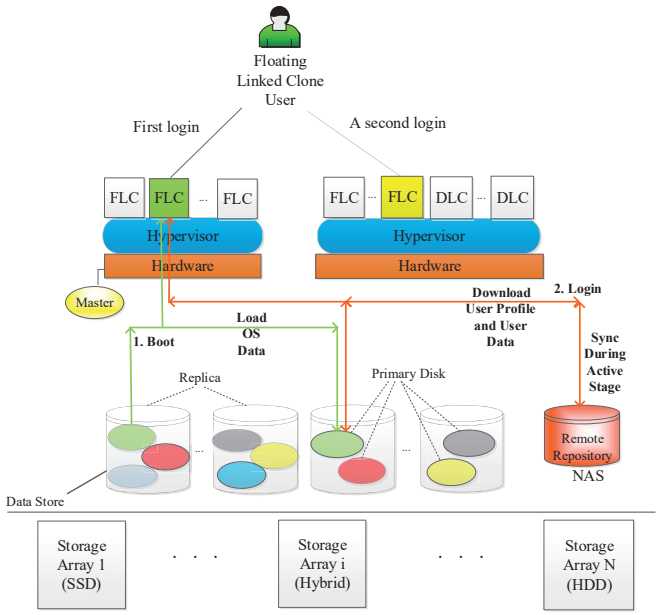


Figure 2.2: Floating Linked Clone Storage Configuration.

virtual desktop is booting, shared OS data have to be read from the replica first. These data are loaded into VM memory to initiate a system boot. Those essential binaries, libraries, etc. are written to the linked clone primary disk as well for future accesses. When a user tries to log in, the virtual desktop must first load user profiles from the remote repository to memory to authenticate the user and then configure the desktop settings. The user profiles are also written to the linked clone's primary disk for future accesses. After the desktop environment is loaded, the virtual desktop goes to the active stage. The user operations during active stage may need to access user data like the user's personal documents, videos, photos, music, etc. stored in the remote repository. These data are downloaded to the primary disk when first accessed. All subsequent accesses are directed to the copies on primary disk. Any changes to the user data are synchronized to the remote repository at regular intervals. Once the user logs off, that virtual desktop is cleaned, so no user profiles or user data is saved on that primary disk. When that user logs into his/her desktop again, a different VM may be assigned.

Data accesses of dedicated linked clones. The data accesses of dedicated linked clones are shown in Figure 2.3. During the boot process, there is no need to load OS

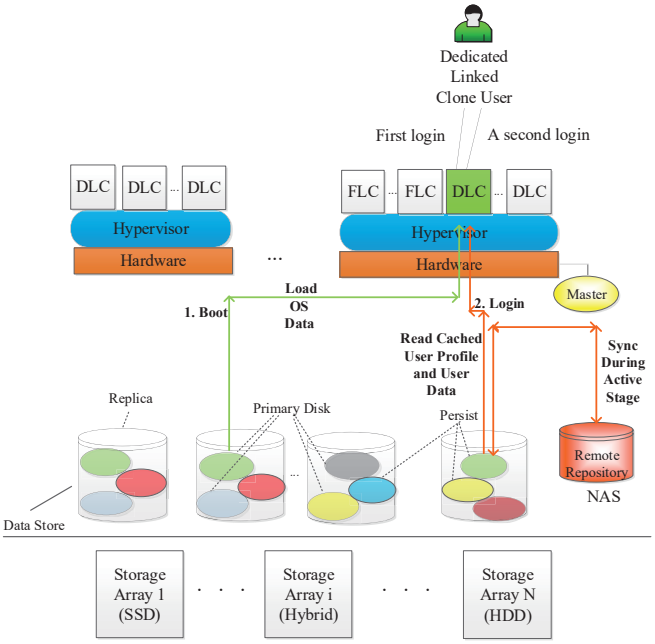


Figure 2.3: Dedicated Linked Clone Storage Configuration.

Table 2.1: Virtual Disks Accessed at Each Stage by Different Virtual Desktop Types
B=Boot L=Login A=Active

	Replica	Primary Disk	Persistent Disk	NAS	Full Clone Disk
Floating Linked Clone	B	B,L,A	-	L,A	-
Dedicated Linked Clone	-	B,L,A	L,A	A	-
Full Clone	-	-	-	-	B,L,A

data from the replica anymore as long as it is not the first boot of a fresh desktop. Those OS data are already stored in primary disk. During the login process and active stage, user profiles and user data are read from the persistent disk rather than from the remote repository. The persistent disk acts as a cache of the remote repository. During active stage, synchronization of user profiles and user data between persistent disk and the remote repository occurs.

Data accesses of full clones. A full clone is like a regular desktop. All information including OS data, user profiles, and user data is stored in full clone disk. Thus, all I/O accesses are on this type of virtual disk during all stages.

Table 2.1 shows when each type of virtual desktop will access each kind of virtual disk. In the table, we omit master image because it is not accessed by virtual desktops

at runtime. We also omit logoff stage because people do not care about the performance of logoff.

2.3.3 VM Model

We define a model to answer at time t, how much data will be read from and written to each virtual disk at a given time t. The basic idea is to sum all read I/Os and write I/Os happening on the same virtual disk at time t. In the following subsections, we discuss the model for a single virtual desktop and multiple virtual desktops, respectively.

Single VM

Formula 2.1 calculates the overall amount of data accessed on a *target* by a VM in life cycle *stage* for a single VM. The *target* is the virtual disk that I/Os reach as listed in Table 2.1. The *stage* is the VM life cycle. $RWper_{\text{stage,target}}$ is the read ratio or write ratio during different stages on different targets when we calculate the size of read data and write data, respectively. The I/O sizes $S^i_{\text{stage,target}}$ are several discrete values. Since there are many different I/O sizes, here we only choose several significant I/O sizes at each life cycle stage on each target. An I/O size is significant when it accounts for most of the I/Os. We decide significance using two factors: 1) the access frequency of the I/O size is high, and 2) the total amount of data transferred under this I/O size is large. For a stage and target, the percentage of total I/Os that are of a certain significant I/O size i is denoted as $Psize^i_{\text{stage,target}}$. $E_{\text{stage,target}}(t)$ describes the expected number of I/Os at time t, which tells how many I/Os arrive at *target* during *stage* at time t. In practice, when calculating how many I/Os are expected to come at time t, we can multiply by a small time interval dt (e.g., 1 second). We do the summation for all significant I/O sizes i to obtain the overall amount of data accessed on a target at VM life cycle stage.

$$\sum_i E_{\text{stage,target}}(t) \times dt \times RWper_{\text{stage,target}} \times S^i_{\text{stage,target}} \times Psize^i_{\text{stage,target}} \tag{2.1}$$

Multiple VMs

In contrast to a single VM, more factors need to be considered when including a number of virtual desktops: 1) VMs start to boot (arrive) at different time, 2) I/O behavior of different virtual desktop types are different. 3) IO behavior of VMs running different operating systems or user applications are different. The VM arrival rate is the key to including multiple VMs in the model. In a data center, VMs arrive at different time, so the multiple VM model should also include an arrival distribution. We use a function $N(x)$ to describe the number of VMs arriving at time x. Different virtual desktop types determine which virtual disks are targets. Virtual desktop types along with the operating system types and user applications affect the arriving I/O sizes as well as their percentages at time t. When we determine the amount of data accessed on each target, we base it on the virtual desktop type and the stage they are in according to Table 2.1.

Assumptions In order to make the model of multiple VMs simple and practical, we make several assumptions. First, the VM arrival rate is not related to virtual desktop types, operating systems, or user applications, and this arrival rate of different virtual desktops follows a distribution. Second, login immediately follows boot and there are no idle intervals. In the following subsections, we show how these assumptions are applied to the model.

Multiple VMs of the Same Type If multiple VMs are of the same virtual desktop type and have the same operating system and user applications, their parameters are all the same. In this case, we only need to consider how to integrate the I/O requests of VMs at different stages into the model. The overall amount of data accessed on virtual disk *target* by VMs at life cycle *stage* for multiple VMs of the same type can be calculated using Formula 2.2. $N(x)$ is the VM arrival rate and indicates the number of VMs arriving at time $x(wherex \leq t)$. For each group of $N(x)$ VMs that arrive at time x, $E_{\text{stage,target}}(t)$ describes the expected number of I/Os for these VMs at time t. In other words, it tells how many I/Os arrive at *target* from VMs in *stage* at time t. For all VMs that are now in *stage* at time t, their arrival time must fall into a prior time interval $[t_1, t_2]$ determined by the amount of time needed for each stage and where

$t_2 \leq t$. We calculate how much data is read or written by VMs currently in *stage* that arrived at any time point in $[t_1, t_2]$ and add them together to obtain the overall amount of data accessed on virtual disk *target* by VMs in life cycle *stage* at a given time t. The other parameters are the same as the single VM model.

$$\sum_{x=t_1}^{t_2}[N(x) \times \sum_i (E_{\text{stage,target}}(t) \times dt \times RWper_{\text{stage,target}} \times S^i_{\text{stage,target}} \times Psize^i_{\text{stage,target}})] \tag{2.2}$$

Here we take one virtual disk as an example. Suppose the target is a primary disk, and we want to calculate the amount of data read from this target at time t. According to Table 2.1, linked clones access a primary disk in the boot, login, and active stages. Therefore, the data accesses should be the combination of three parts: I/Os from VMs in the boot process, I/Os from VMs in the login process, and I/Os from VMs in active stage. I/Os from VMs in the boot process at time t can be calculated by

$$\sum_{x=t-t_0}^{t}[N(x) \times \sum_i (E_{\text{boot,primary}}(t) \times dt \times RWper_{\text{boot,primary}} \times S^i_{\text{boot,primary}} \times Psize^i_{\text{boot,primary}})] \tag{2.3}$$

Here we assume each of these VMs of the same type takes t_0 time units (e.g., seconds) to finish issuing I/Os during its entire boot process. VMs that started booting (arrived) during time interval $[t - t_0, t]$ are still in their boot process. For every group of VMs that arrived at each time point in $[t - t_0, t]$, we calculate how much boot data they read from the virtual disk at time t and add them together. Similarly, I/Os from VMs in the login process at time t can be calculated by

$$\sum_{x=t-t_0-t_1}^{t-t_0}[N(x) \times \sum_i (E_{\text{login,primary}}(t) \times dt \times RWper_{\text{login,primary}} \times S^i_{\text{login,primary}} \times Psize^i_{\text{login,primary}})] \tag{2.4}$$

Here we assume VMs of this same type take t_1 time units to finish issuing I/Os during their entire login process. According to Assumption 2, login follows boot immediately, and there are no idle intervals. Therefore, VMs that arrived during time interval $[t - t_0 - t_1, t - t_0]$ are in their login process at time t. Finally, I/Os from VMs in their active stage at time t can be calculated using

$$\sum_{x=start}^{t-t_0-t_1} [N(x) \times \sum_{i} (E_{\text{active,primary}}(t) \times dt \times RWper_{\text{active,primary}} \times S^{i}_{\text{active,primary}} \times Psize^{i}_{\text{active,primary}})] \tag{2.5}$$

Here we assume an initial point in time *start* when VMs start to arrive. Any VMs that arrived before $t - t_0 - t_1$ are now (at time t) in their active stage.

Since a virtual disk is not accessed during all stages by all virtual desktops, when calculating the total amount of data read/written on target at time t, we include only I/Os from appropriate virtual desktop types and the stages they are in according to Table 2.1. By selecting different targets, we can obtain the amount of data accessed on all virtual disks. By traversing time t, we can see how data accessed on a target varies with time. Therefore, we can estimate when a bottleneck happens on each target.

Multiple VMs of Different Types VMs with different virtual desktop types, operating systems, and user applications show different I/O behavior. We define a VM type as VMs running the same type of virtual desktop, the same type of operating system, and the same user applications. For each VM type, we apply Formula (2.2) to calculate how much data are read from and written to each corresponding target at time t. The corresponding targets are chosen from Table 2.1 according to the virtual desktop type. In a data center, each VM type accounts for a different proportion of I/Os. When combining them together, we need to use the weighted proportion of each VM type. According to Assumption 1, the VM arrival rate is not related to its virtual desktop type, operating system, and user application. The overall arrival rate *N(x)* is the same when calculating each VM type. Therefore, we can use the weighted average of all VM types to get the total amount of data accessed.

2.4 Data Analysis and Evaluation

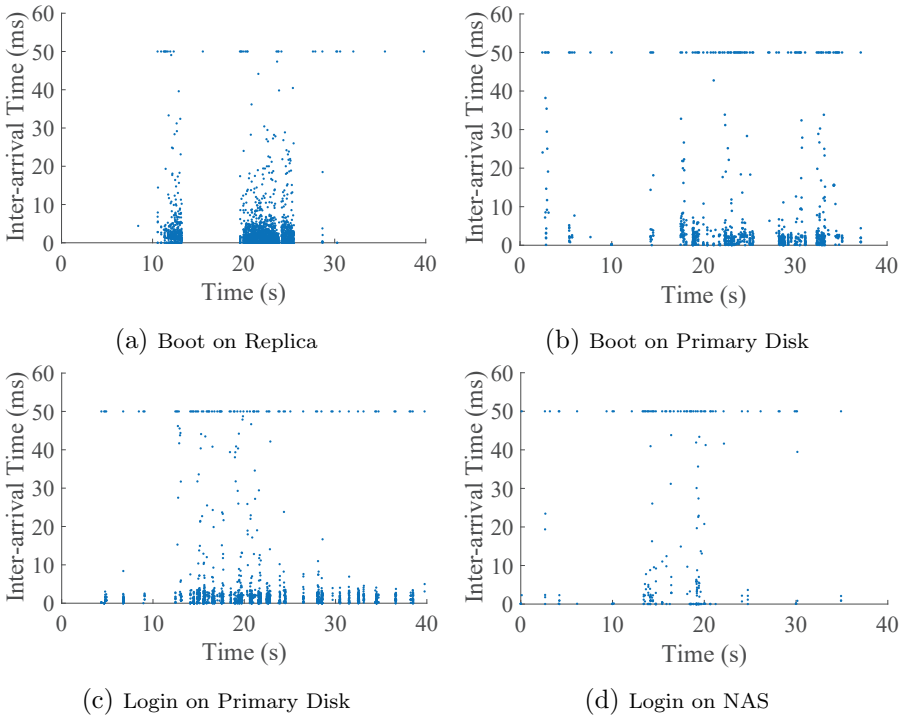
In order to get correct values of I/O parameters in our model, we collect boot, login, and active stage traces of different types of virtual desktops in VDI. We then analyze I/O behavior of virtual desktops and derive those parameters in our model from the traces. The storage demands are then generated.

In this section, we first analyze the burstiness of I/Os in order to describe the expected number of I/Os at time t in our model. Then we analyze I/O behavior of each type of virtual desktop from traces. Finally, we show a simulation using our model to generate I/O demands on each target.

2.4.1 Trace Collection

We collect VDI traces on four 1U Dell r420 servers, each with two Intel Xeon E5-2407 v1 2.2GHz quad-core processors and 12 GB of DRAM. Servers are connected to Dell/Compellent SC8000 storage which has eight 400GB SSDs and 8 600GB HDDs. In our trace capturing VDI environment, we have VMware vSphere Hypervisor (ESXi) 5.5.0 installed on four servers to form a cluster. We use vCenter Server 5.5 Update 1 as the cluster manager. The VDI product installed is VMware Horizon View 6.0. Windows 7 x64 is used as the desktop OS. To capture traces of different clone types, we set up floating assigned linked clone pools, dedicated assigned linked clone pools and full clone pools. In this environment, any I/Os generated by a virtual desktop go through the hypervisor (ESXi) first and then go to the physical storage. To capture block traces, a common method is running blktrace at the host machine. Unfortunately, ESXi is a commodity hypervisor so we cannot install or run blktrace on it. Instead, we use a different method. We setup an NFS server to provide storage to the virtual desktops. We mount multiple volumes on this NFS server and configure them as data stores of ESXi servers. Through this configuration, we can choose to put virtual disks in the data stores presented by this NFS server when creating virtual desktop pools. At the NFS side, we run blktrace on multiple volumes to collect block traces. By analyzing the block traces collected from different volumes, we are able to analyze the I/O behavior of different virtual desktops.

Since NFS just transmits the original file I/Os to VMDK (Virtual Machine Disk)



(a) Boot on Replica

(b) Boot on Primary Disk

(c) Login on Primary Disk

(d) Login on NAS

Figure 2.4: Inter-arrival Time of I/Os of a Floating Linked Clone

from ESXi hosts to the NFS server, the I/O patterns in traces collected from ESXi (if we could) should be the same as those in traces collected from our NFS server. Using this setup, we collect traces of virtual desktops at their boot and login stages. During the active stage, the workload depends on the real user application behavior. Here we run VMware View Planner 3.5 [76] as a workload generator to generate active stage workload. The applications include Adobe Reader, Excel, Internet Explorer, PowerPoint, and Word. VMware View Planner executes open, read, write, save, and close operations using these applications to simulate a real real-life workload. We set the workload to pause randomly for a duration between 0 and 10 seconds between operations, which is a common configuration of View Planner to emulate human activities on desktop. In this way, we generate light load during an active stage.

2.4.2 Burstiness of Requests

Figure 2.4 depicts the inter-arrival time of I/Os of a floating linked clone from the traces. For those sparse requests whose inter-arrival time is greater than 50ms, we simply truncate them to 50ms. The coefficients of variation of the inter-arrival time from (a) to (d) in Fig. 2.4 are 13.61, 8.15, 8.72, and 3.57 respectively. A higher coefficient of variation indicates more bursty arrival patterns. In addition, it is also obvious that data points are aggregating at some time points rather than evenly distributing across time. This also indicates that I/Os on all targets are bursty. Other clone types show similar bursty features. Thus, it is not appropriate to describe the expected number of I/Os at time t by statistical models, like the famous Poisson Process. Since the actual arrival pattern happens during a very short time interval (μs) and we are only interested in measuring I/Os in terms of seconds, we can simply assume that requests arrive uniformly within bursts and plug bursts into our model. The expected number of I/Os at time t, if t falls into a burst, is the average number of I/Os per time unit during the interval of that burst. Otherwise, when t is outside of a burst, the expected I/O count is zero.

2.4.3 Single Virtual Desktop Analysis

In this subsection, we analyze the I/O behavior of a single virtual desktop from traces. The traces include the I/O requests sent to different targets by different types of virtual desktops in boot, login, and active stages.

Figure 2.5 shows an I/O workload during active stage generated by VMware View Planner. In our trace, the VMware View Planner pauses for a random time between 0 to 10 seconds in the middle of open, read, write, save, and close operations. It accesses multiple applications. A simple read or write operation on different applications (e.g., GET a web page vs. read a pdf) will generate different amount of data. Accessing different files will cause different amount of data being read or written. Due to the existence of caches at different levels, some of the operations may not generate I/Os to the block device. With these factors, the amount of data being read and written during the active stage shows a random pattern. In this research, we aim to identify the storage requirements of VDI and determine the bottlenecks on virtual disks. Since

the I/O workload during active stage is less intensive than those of boot and login stages, the bottlenecks mainly happen during boot and login stages. Therefore, we can model the expected amount of data being read or written during active sub-stages with variations based on statistical averages. In the trace, we use only one configuration of the VMware View Planner, such that we can model the active stage with a single stage.

In this subsection, we will focus our discussion on boot and login stages. The I/Os in these stages are more critical. They can form an I/O storm and influence the overall performance very much.

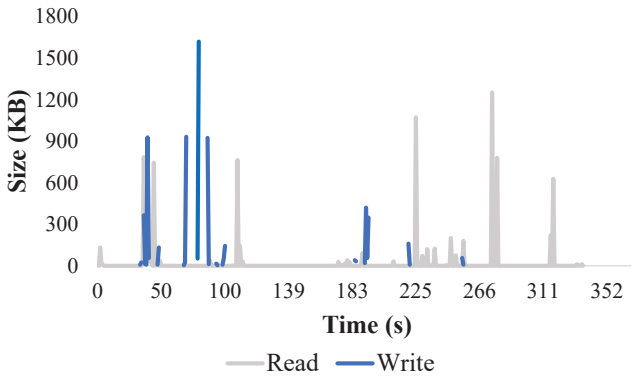


Figure 2.5: Amount of Data Accessed During Active Stage

Floating linked clone

During the boot process of a floating linked clone, reads are dominant on the replica, accounting for 99.8% of total I/Os on the replica. The total amount of data read is 188MB. To the contrary, writes become dominant on the primary disk, accounting for 99.9% of total I/Os on the primary disk. The total amount of data written is 20MB. This is because when a floating linked clone is booting, it needs to load OS data from the shared replica first. Some of this data is written into the primary disk for future use. During the login process, I/Os happen on the primary disk and NAS. On the primary disk, reads account for 22.7% and writes account for 77.3% of disk activity. On the NAS, reads account for 69.5% and writes account for 30.5% of I/Os. In this process, the user profile needs to be loaded from NAS to enable identity authentication and desktop configuration. Some of these data are written into primary disk for future

use. Other applications involved during the login process may also write to the primary disk.

We obtain the detailed storage requirements of a floating linked clone as listed in Table 2.2. The throughput and IOPS are only measured when there are I/Os going to the virtual disks in certain stages. For example, there are no write I/Os on replica during boot stage of a floating linked clone. We do not count the boot stage into the write throughput or write IOPS requirement on replica.

Table 2.2: Requirements of a Floating Linked Clone

	Replica		Primary		NAS	
	R	W	R	W	R	W
Average KB/s	4813.90	0	279.91	641.98	212.91	15.66
IOPS	68.55	0	12.26	29.41	5.00	2.14
Capacity per VDI user if thin provisioned: 4GB. 3 shared replicas: 75GB Shared NAS: 1TB						

Dedicated linked clone

The boot process of a dedicated linked clone is quite different from a floating linked clone. Dedicated linked clones preserve data in primary disks after users log off rather than reload data for every instance as is done with floating linked clones. In most cases, dedicated linked clones do not need to load OS data again from a replica during the boot process. Loading OS data into the primary disk only happens the first time a fresh desktop is booted. Here we only consider the most general case where primary disks already preserve OS data. In the traces, we find boot I/Os only go to primary disks. Dedicated linked clones utilize persistent disks to preserve user profiles and user data. Therefore, I/O behavior of dedicated linked clones during login are different from floating linked clones. Some I/Os are now shifted to the persistent disk, and the number of I/Os accessing NAS is reduced. This is because we do not need to load as much data from NAS to the primary disk when users log in, since the data are already there in persistent disk. We can directly read user profiles from the persistent disk to proceed with the login process. On the primary disk, reads account for 48.9% and writes account for 51.1%. On the persistent disk, reads account for 24.6% and writes account for 75.4%.

On the NAS, reads account for 32.2% and writes account for 67.8%.

Detailed storage requirements of a dedicated linked clone are listed in Table 2.3.

Table 2.3: Requirements of a Dedicated Linked Clone

	Primary		Persistent		NAS	
	R	W	R	W	R	W
Average KB/s	269.24	397.87	49.68	31.39	27.13	12.87
IOPS	8.36	22.60	1.65	2.46	0.82	1.78
Capacity per VDI user if thin provisioned: 12GB. 3 shared replicas: 75GB Shared NAS: 1TB						

Full clone

In a full clone, I/Os are aggregated in one type of virtual disk. During the boot process, reads account for 42.2% and writes account for 57.8%. During the login process, reads account for 69.0% and writes account for 31.0% of I/Os.

Detailed storage requirements of a full clone are listed in Table 2.4.

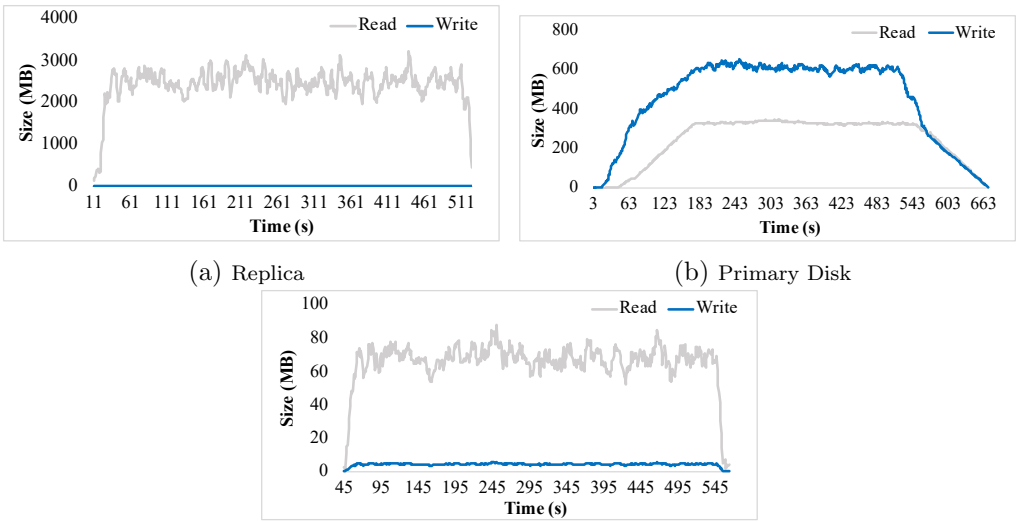
Table 2.4: Requirements of a Full Clone

	Full Clone Disk	
	R	W
Average KB/s	1401.54	284.09
Average IOPS	26.34	22.18
Capacity per VDI user if thin provisioned: 12GB.		

2.4.4 Multiple Virtual Desktops

With traces collected for each type of virtual desktop, we can now aggregate multiple virtual desktops together. We do experiments to simulate multiple virtual desktops arriving at different times and see how the I/Os on each target vary with time.

Experiment Setup. We assume a company uses VDI for its employees and has 5,000 virtual desktop instances. Without loss of generality, we assume the arrivals of employees follow a Poisson distribution and the arrival rate is 10 per second. In order to aggregate I/Os of VMs at different stages in Formula 2, we use the parameters derived



(a) Replica (b) Primary Disk

(c) NAS

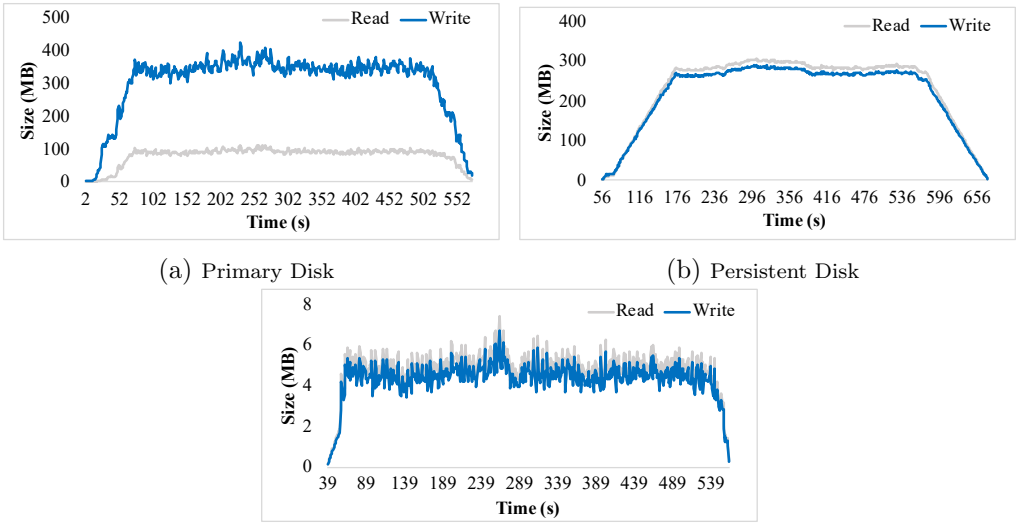
Figure 2.6: Amount of Data Accessed on Targets of 5,000 Floating Linked Clones

from the traces in the model. Other users of our model can add their own customized virtual desktop types and VM arrival rate to get their own results. In the following subsections, we show how much data is accessed on each virtual disk at any time since the first user arrives under four scenarios: 1) they uses all floating linked clones, 2) they use all dedicated linked clones, 3) they use all full clones, 4) they use a mixture of different clones.

Floating linked clones

If we assume this company prefers that employees share virtual desktops as much as possible in order to reduce license costs (Windows, VMware, etc.), it may use all floating linked clones. Figure 2.6 shows the amount of data accessed on each of the targets from when the first floating linked clone arrives to when all floating linked clones have transitioned to active stage.

On the replica, as seen in Figure 2.6(a) the I/Os are read dominant and quite heavy. The rate of data read rises sharply to around 2.8 GB/s within the first 30 seconds. In the next 500 seconds, the workload is relatively stable and stays around 3 GB/s with a peak of 3.3 GB/s. Once all virtual desktops have arrived and their boot processes



(a) Primary Disk

(b) Persistent Disk

(c) NAS

Figure 2.7: Amount of Data Accessed on Targets of 5,000 Dedicated Linked Clones

are completed, the I/Os start to drop dramatically within the final 20 seconds. Overall, replica disk activity is read intensive for floating linked clones. Unlike the replica, the I/Os on the primary disk in Figure 2.6(b) are more balanced as it is accessed during all stages. However, it still sees a large volume of I/Os with data rates in the hundreds of megabytes per second for reads and writes. As shown in Figure 2.6(c), the amount of data accessed on NAS is quite small. If more applications are installed and more user data is generated, NAS activity will increase.

Dedicated linked clones

If we assume this company wants to reduce license costs and at the same time employees are inclined to have dedicated virtual desktops, it may use all dedicated linked clones. Figure 2.7 shows the amount of data accessed on each of the targets from when the first dedicated linked clone arrives until all dedicated linked clones have transitioned to active stage.

The I/Os on the primary disk of the dedicated linked clones, as seen in Figure 2.7(a), are much lighter than those of the floating linked clone. Once all VMs finish their boot and login stages, the I/Os on the primary disk are minimal. On the persistent disk, as

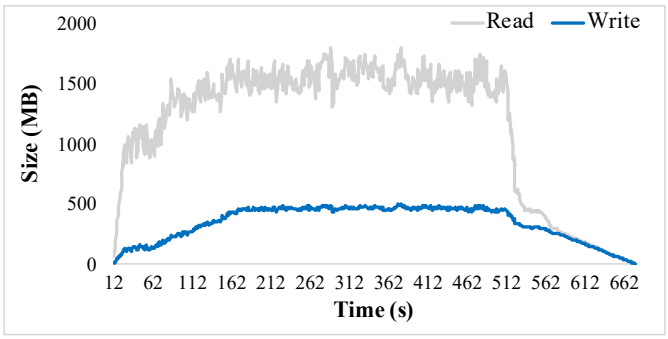


Figure 2.8: Amount of Data Accessed on Target of 5,000 Full Clones

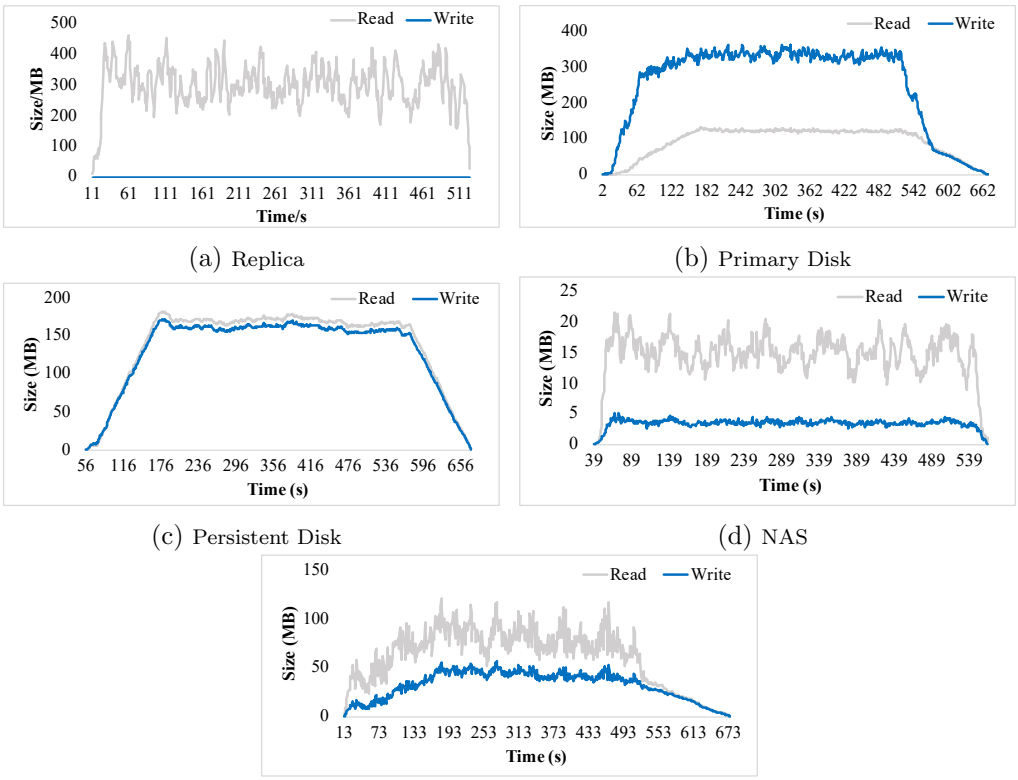
we see in Figure 2.7(b), reads and writes mainly rise during the login stage and drop to a minimum amount in the active stage. Figure 2.7(c) shows I/Os on NAS.

Full clones

If we assume this company wants to avoid the complexity of server and storage configurations caused by linked clones, it may use all full clones. Figure 2.8 shows the amount of data read and written on the full clone disk. We can see the total amount of data read is much greater than the total amount of data written. There is an obvious stage of high I/Os where VMs are in their boot and login stage. The I/Os drop suddenly when all VMs finish booting and then gradually decline to the minimum as the VMs transition to active stage.

A mixture of different clones

Now assume this company has various needs regarding virtual desktops. First of all, it does not want to pay high license costs, hence most virtual desktops are linked clones. Also, most employees require exclusive use of virtual desktops, so most of the linked clones are dedicated linked clones. Finally, there is a small department developing software tools that require high performance. To avoid the configuration complexity and degraded performance of linked clones, a small number of full clones are provided. According to this scenario, we evaluate a combined ratio of 3:6:1 for floating linked clones to dedicated linked clones to full clones. Figure 2.9 shows the amount of data read and written on the replica, primary disk, persistent disk, NAS, and full clone



(a) Replica

(b) Primary Disk

(c) Persistent Disk

(d) NAS

(e) Full Clone Virtual Disk

Figure 2.9: Amount of Data Accessed on Targets of a Mixture of Clones

virtual disk in this mixed configuration.

The I/Os on the replica arise from the boot of floating linked clones. The amount of data accessed shows a similar pattern as Figure 2.6(a). Since the number of floating linked clones is only 30% of the 5,000 virtual desktops, there are fewer I/Os than there were in the 5,000 floating linked clone case. Compared with dedicated linked clones (Figure 2.7(a)), floating linked clones (Figure 2.6(a)) have more intensive I/O accesses on the primary disk. The pattern of data accessed on primary disk in this mixed case (Figure 2.9(b)) is similar to the floating linked clone case. However, the total amount of data accessed is smaller. The persistent disk (Figure 2.9(c))is exclusively accessed by the dedicated linked clones, so its I/O pattern is similar to Figure 2.7(b). Although dedicated linked clones access NAS (Figure 2.9(d)), the I/Os are reduced a lot due to the existence of the persistent disk. Therefore, the I/O pattern on NAS is similar to

Table 2.5: Comparison Between the Calculated Throughput Requirements with the Measured Throughput Requirements

Virtual Desktop	Virtual Disk	Average Read KB/s			Average Write KB/s		
		Measured	Calculated	Diff	Measured	Calculated	Diff
Floating Linked Clone	Replica	4813.90	4812.14	0.04%	0	0	0
	Primary	279.91	277.03	1.03%	641.98	649.62	1.19%
	NAS	212.91	209.33	1.68%	15.66	15.05	3.90%
Dedicated Linked Clone	Primary	269.24	266.98	0.84%	397.87	397.62	0.06%
	Persistent	49.68	48.85	1.7%	31.39	30.85	1.7%
	NAS	27.13	25.97	4.3%	12.87	12.74	1.0%
Full Clone	Full Clone Disk	1401.54	1400.66	0.06%	284.09	282.74	0.48%

that of the floating linked clones (Figure 2.6(c)) but with a smaller total data amount. Finally, the full clone virtual disk (Figure 2.9(e)) is only influenced by full clones, so it shows the same pattern as in Figure 2.8 with a proportional reduction in data trasferred.

2.4.5 Validation

We evaluate the correctness of our model from two aspects. We first compare the throughput requirement calculated from our model of a single VM (Equation 2.1) with the direct measurement as described in Section 2.4.3. Then, we compare the parameters calculated from the simulation according to the model of multiple VMs (Equation 2.2) with the experimental results from two Hewlett Packard Enterprise (HPE) systems.

We first use the parameter values derived from the traces into the model of a single VM and calculate the average size of data accessed on each virtual disk for each type of virtual desktop. In Table 2.5, we summarize the comparison between the throughput requirements calculated from our model and the throughput requirements measured in Section 2.4.3. Table 2.5 shows that our model can generate the average size of data read and write per second on each virtual disk with small errors, as indicated by read diff and write diff. When calculating the average read or average write, we do not count a stage if there are no read or write I/Os in that stage.

Next, we evaluate the case when there are multiple virtual desktops running in a VDI environment. We generate the I/O requirements from the simulation by plugging the traces from Section 2.4.1. We also carry out an experiment on two different real Hewlett Packard Enterprise (HPE) systems which run IOmark-VDI [77] to generate VDI workloads. We compare the I/O requirements from these two experiments. The

first HPE system has three HP Proliant BL460c Gen8 servers. Each server has two Intel Xeon 2.9GHz, octa-core CPUs and 128GB of memory. One HPE 3PAR StoreServ 7450 is used as VDI storage. It has 24 920GB MLC SSDs and 8Gbps Fibre Channel SAN connectivity. 40 volumes are provisioned for testing, each of which has 125GB of capacity. VMware vSphere 5.1 and vCenter Server 5.1 are the deployed hypervisor and cluster manager. The second system has two HP ProLiant DL560 Gen8 servers. Each server has four Intel Xeon 2.7GHz octa-core CPUs and 256GB of memory. One HPE StoreVirtual 4335 is used as VDI storage. It has nine 400GB MLC SSDs, 21 900GB 10K RPM SAS HDDs, and 10Gbps iSCSI SAN connectivity. 12 volumes are provisioned for testing, each of which has 125GB of capacity. VMware vSphere 5.0 and vCenter Server 5.0 are the deployed hypervisor and cluster manager.

In both HPE systems, IOmark-VDI [77] is used as the benchmark tool to generate VDI workloads. This tool re-creates a VDI environment automatically. The virtual desktops created are running Windows 7 as the guest OS. A boot storm of floating linked clones is created by IOmark-VDI. Each desktop has various applications installed including Word, Excel, PowerPoint, Outlook, Internet Explorer, Adobe Reader, etc. Since IOmark-VDI does not provide a login stage workload, we only compare the experimental results from these two HPE systems with our simulated results in the boot stage and active stage.

The experimental results from two HPE systems are shown in Tables 2.6 and 2.7, respectively. In the experiment, different numbers of floating linked clones are booted at the same time. During active stage, the IOmark-VDI is running in "Standard" mode which generates light workload. From these tables, we can see the peak IOPS per virtual desktop, which occurs during the boot stage, is 139 regardless of the testing environment as long as there are enough system resources. The peak read IOPS is around nine times of the peak write IOPS. During active stage, the average IOPS is 6.26. To evaluate our model, we simulate multiple floating linked clones arriving at the same time. We set the arrival rate of virtual desktops to match the total number of virtual desktops from the experiment on the HPE systems. For example, for user count of 512, we set *N(x)* equal to 512 when x is 1. Otherwise, *N(x)* equals to 0. Each floating linked clone will generate $E_{\text{stage,target}}(t)$ number of I/Os at time t. The value of $E_{\text{stage,target}}(t)$ is calculated as the average number of I/Os per second during the burst, as described in Section 2.4.2. We

Table 2.6: I/Os on HP 3PAR StoreServ 7450

User Count	Max IOPS	Max IOPS per user	Max Reads per second	Max Writes per second
512	71168	139	64051	7117
1056	146784	139	132195	14589
1504	209056	139	188150	20906
2016	280224	139	252202	28022

Table 2.7: I/Os on HP StoreVirtual 4335

User Count	Max IOPS	Max IOPS per user	Max Reads per second	Max Writes per second
148	20513	139	18462	2051
211	29329	139	26396	2932
318	44202	139	39781	4420
537	74643	139	67178	7465

record down the IOPS on each virtual disk changing with time. We find the peak IOPS per floating linked clone on all virtual disks is 141 and the peak read IOPS is 8.8 times the peak write IOPS. The peak IOPS happens during the boot stage. During active stage, the average IOPS is 5.29. These results show that the simulation based on our model is able to represent the real workload characteristics from the HPE experiment.

2.5 Application of Proposed Model

In this section, we show how to apply our model in real life. We identify more fine-grained storage requirements of a VDI system and compare them with the storage performance requirements provided by VMware. Then we show what storage system is needed to deploy such a VDI system.

2.5.1 Fine-grained Storage Requirements

We first show we can find more accurate and fine-grained storage requirements of a VDI system.

To assist administrators when they are determining the resources necessary to support VDI, VMware has given IOPS requirements as a rule of thumb [78] as shown in

Table 2.8: VDI IOPS Requirements from VMware

User Classification	IOPS Requirements Per User
Light	3-7
Medium	8-16
Standard	17-25
Heavy	25+

Table 2.9: Specifications of 4 HP 3PAR Storage Systems

HP 3PAR Storage	F200	F400	T400	T800
Max Throughput	1300 MB/s	2600 MB/s	2800 MB/s	5600 MB/s
Max IOPS	46,800	93,600	128,000	256,000
Max Capacity	128 TB	384 TB	400 TB	800 TB
Drive Types	50GB SSD 300 & 600 GB FC 2TB NL	50GB SSD 300 & 600 GB FC 2TB NL	50GB SSD 300 & 600 GB FC 2TB NL	50GB SSD 300 & 600 GB FC 2TB NL

Table 2.8. It classifies users based on their IOPS requirements. However, we know different types of virtual desktops have different storage requirements, and for the same type of virtual desktop, the storage requirement of each virtual disk is also different. At different stages during the life cycle of a virtual desktop, the storage requirements also change. The VMware guidance based on the rule of thumb does not describe these differences in detail. In Tables 2.2, 2.3, and 2.4 in Subsection 2.4.3, we show the detailed read, write IOPS requirements on each virtual disk for each type of virtual desktop at boot and login stages. In our VDI traces, we use VMware View Planner to generate light workload, including reading PDF files, editing Word, Excel, and PowerPoint documents, surfing internet through Internet Explorer, etc. The workload yields an average IOPS of 5.29. It matches the IOPS requirement range of the *Light* user. However, at other stages, the IOPS requirements are quite different. For floating linked clone, the read IOPS requirement on a replica can be 68.55, more than 12 times of the IOPS requirement during the active stage. However, the rule of thumb misses the complexity of IOPS requirements inside VDI. It can only give guidance on the IOPS requirements during active stage.

IOPS is widely used in industry to describe storage requirements and capabilities. VMware uses IOPS to guide VDI storage sizing. For example, they use the IOPS in Table 2.8 to calculate the performance requirement for each LUN (logical unit number, used to identify a device or a logical disk) when determining the storage hardware necessary for VDI. However, only considering IOPS is less than adequate. As seen in

Section 2.4, the amount of data read per second from the replica can be very large during boot time. However, the IOPS requirement on replica listed in Table 2.2 does not show this bandwidth requirement directly. If a user allocates storage to VDI simply based on the IOPS requirements, he may not assign the correct storage. Thus, throughput and read/write ratio should also be considered when allocating storage for VDI. Fortunately, our model can provide this type of information and can guide administrators to an even more fine-grained storage configuration. Our model can show the storage requirements like storage capacity and throughput on each target and how they vary with time directly. In addition, some VDI users have response time requirements. We can find the expected response time of various storage systems by combining the expected IOPS from our VDI model with the Response Time/Throughput relationship shown in benchmark results provided by the SPC (Storage Performance Council, they have extensively tested many storage systems and freely provide plots of response time given IOPS for each system) [79, 80].

2.5.2 Sizing Storage Hardware for Specific VDI Requirements

When we decide how much storage hardware we need, we can first look at the VMware guidance [69] which considers IOPS during active stage and storage capacity. Then we add more dimensions by considering the distinct I/O access patterns in terms of read/write ratio and throughput on different types of virtual disks at different stages during the VM life cycle. For example, assume we are sizing storage for a company that uses VDI for its employees across three time zones. In each time zone, it deploys 5,000 floating linked clones and the I/O access patterns of these virtual desktops are the same as the example in Section 2.4.4. We are going to buy one of the storage systems from Table 2.9. The prices of the storage systems in the table increase as the performance improves. We are going to choose the cheapest storage system that can meet the storage requirements of the VDI system.

We can first consider the I/Os during active stage, as it is only determined by user behavior. If employees in this company generate a light amount of IOPS, e.g. 7 IOPS per user, then we need a storage system that can support $7 \times 5000 \times 3 = 105,000$ IOPS. In this case, HP 3PAR T400 may be a good choice. Then we should consider the I/Os during the boot and login stage, which are less user related and more determined by

the virtual desktop type itself. Since these 5,000 virtual desktops finishes booting and login within a small time period according to Section 2.4.4, I/Os during boot and login stages from different time zones do not overlap. But they occur periodically. According to our model, as described in Section 2.4, we know the rate of data read on replica will keep around 3 GB/s and rise to 3.3 GB/s at maximum. The primary disk will see a stable read rate of 350 MB/s and stable writes at 600 MB/s. The remote repository will stay at 70 MB/s of reads. We consider the most intensive I/O access period here in order to give the best guarantee. In total, this company will see approximately 4 GB/s of sustained data access periodically. The total capacity requirement can also be calculated to be 66 TB. Therefore, we need a storage system that has at least 4 GB/s of bandwidth and 66 TB of capacity. In this case, HP 3PAR T400 cannot meet this throughput requirement, and the HP 3PAR T800 can be a good choice.

If we inspect the I/O throughput on each virtual disk, we can find most of the load is on the replica, and the throughput requirement on the primary disk and remote repository is not that high. Therefore, we can deploy different targets on different storage, and we suggest using tiered storage to satisfy the storage requirements with minimum cost. We know there is a high throughput requirement on replicas, and I/Os are read dominant, so it is a perfect match to deploy replicas on SSDs. We call this group of SSDs Tier-1 storage. Compared with the replica, I/Os on the primary disk are more balanced. Considering that the workload is more write intensive on the primary disk, SSDs do not help as much (assuming SSD writes are slower than SSD reads). To save money, we can place primary disks and the remote repository on HDDs. For better performance, we can use high performance HDDs, e.g., 15K RPM HDDs. We call these HDDs Tier-2 storage. In this case, we can configure an HP 3PAR T800 into tiered storage, which is able to migrate data between tiers automatically to reduce the number of disks needed. Therefore, in order to satisfy the storage requirements of this company, we suggest buying an HP 3PAR T800 and configuring it as tiered storage to reduce cost.

2.6 Discussion and Future Work

Current VDI design is built on virtual machines. Recently, a lightweight virtualization technology call containers [3, 4] has become widely accepted in industry. Applications can run in containers just like in traditional VMs. Containers on the same host will share the same operating system kernel. Each container can have its own libraries, binaries, and namespace. They are segrated on the same host. From the host's point of view, each container runs as a process. On the other hand, VMs run on the hypervisor, a specialized OS, upon which each VM will run a full copy of an operating system. Consider a situation where existing resources cannot meet the storage requirements of VDI (e.g., if the company in the previous section where using an HPE 3PAR F400 to run 5,000 floating linked clones). Instead of suggesting an immediate hardware upgrade, we are exploring a possible remedial solution to migrate virtual desktops from VMs to containers, e.g., Docker containers [61, 60]. We will now present some preliminary results and analysis of this idea.

We first collect traces of a virtual desktop built on a Docker container. Booting such a virtual desktop is the process of creating and running a new Docker container. During the boot stage, a total of 18.09 MB of reads and 7.85 MB of writes are generated to boot the container. The reads are mainly due to loading OS data, the Docker daemon and runC (underlying Docker runtime technology) [81]. The writes during boot mainly come from adding a writable "container layer" on top of underlying image layers. In this virtual desktop implementation, the application is configured to run immediately after the container is up. The application inside this container generates 80.08 MB of reads during boot. It is obvious that reads and writes during the boot stage of a virtual desktop inside a container are far fewer than those of a floating linked clone. This is because booting a container is exactly the process of forking a process on the host. There is no need to load all OS data from replicas as floating linked clones. Writes are only for writing the thin writable container layer. In addition, all read I/Os are eliminated during subsequent boots from that virtual desktop because the OS caches the data, and containers share the system cache with the host.

The login stage of virtual desktops inside containers has similar I/O patterns as in VMs. It reads user profiles to authenticate users and configure desktop settings.

User profiles can be stored in a local Docker union file system or in volumes provided by underlying storage. According to the current virtual desktop implementation in containers, all user profiles are permanently stored. There is no need to first load them from a remote repository.

Therefore, it is worthwhile to consider replacing some of the floating linked clones in VMs with virtual desktops using Docker containers in the situation we are looking at. First, virtual desktops in containers can maintain the flexibility of floating linked clones. In our experiments, it only takes 1.45 seconds for the Docker daemon and runC to finish booting a virtual desktop in a container, while it takes 39.80 seconds to boot a floating linked clone in a VM. This nearly instant boot time makes deleting a virtual desktop after a user logs off and booting another when a user logs in cost far less than the same process using floating linked clones in VMs. Second, during the boot process, there are far fewer reads and writes than when using floating linked clones. If it is not the first time running a virtual desktop, data are already cached, and reads can be eliminated. While containers have advantages over VMs, virtual desktops using containers are not yet as mature as using VMs, and only some independent open source projects can be found. Containers themselves also have security limitations as containers share the same OS. Containers also do not support data persistence as well as VMs do, but storage companies are working on this issue. Docker is open source and under rapid development, so the acceptance of virtual desktops using containers will continue to increase.

2.7 Related Work

2.7.1 VDI and Its Enhancement

Currently, there are multiple VDI solutions such as VMware Horizon View [66], Microsoft Virtual Desktop Infrastructure [67], and Citrix Xen [68]. No matter which solution is chosen, storage performance is a big hurdle. VMware stated that over 70% of performance issues are related to storage. There are multiple storage solutions aiming to improve storage performance for VDI. VMware uses content-based read cache(CBRC)[82] to improve performance by caching common disks in the ESX host server. Unlike VMware CBRC, which restricts cache access to the same host, Infinio

builds a distributed version of host side cache [83]. Another solution from PernixData utilizes server flash to accelerate VDI performance [84].

2.7.2 VM Characterization and Storage Requirements

I/O workload characterization [85, 86, 87, 88, 89] has been an important topic for storage researchers. Traditionally, the I/O workloads are collected from physical servers. Recently, more and more researchers have focused on the uniqueness of VM workloads. Tarasov et al. studies the extent to which virtualization is changing existing NAS workloads [90]. Gulati et al. presents a workload characterization study of three top-tier enterprise applications using the VMware ESX server hypervisor [91]. There are also characterizations based on other workloads including cloud backends. Mishra et al. try to characterize the workload of Google compute clusters [92]. Their goal is to classify workloads in order to determine how to form groups of tasks (workloads) with similar resource demands. Although these studies have some characterization of VM I/O behavior, they do not quantify I/O demands from the perspective of satisfying storage requirements.

Most of the studies trying to provide methods of meeting VM requirements overlook the characteristics of the VM storage requirements. Gulati et al. [93] do study how to improve I/O performance of VMs, but they only use Iometer to generate some workloads, thus cannot fully represent the storage requirements of VMs. Le Thanh Man et al. [94] study how to minimize the number of physical servers needed while ensuring Service Level Agreement (SLA) requirements. They place the virtual desktops whose access patterns have low correlation coefficient on the same servers, so the opportunity for CPU contention in the same servers is low. However, they mainly focus on CPU. The storage in VDI has its own characteristics and configurations, thus requiring a special discussion.

The manuals of commercial VDI products and the underlying storage are based on either rules of thumb to guide storage provisioning [78] or test the performance of their storage array given a fixed number of VDI instances [72]. Some products specifically try to meet VM storage requirements. VMware's vSAN [95] is such a product, and VMware has already integrated it with vSphere [96]. vSAN shows the available storage capabilities and claims they can be used to handle the storage requirements of VMs.

When deploying VMs, administrators choose among existing storage and place VMs into the selected storage. However, such a description of VM storage requirements using storage capabilities can be inaccurate.

Another product that tries to meet VM storage requirements is Common Provisioning Group (CPG) [97] from HPE. It pools underlying physical devices into a unified storage pool called a CPG. VMs can draw resources from CPGs, and volumes are exported as logical unit numbers (LUNs) to hosts. CPG gives a detailed organization of the underlying storage. It tries to meet storage requirements of VMs by organizing underlying storage resources, but not from the VM perspective.

2.8 Conclusion

In this chapter, we create a model to identify the storage requirements of one prevalent virtual machine type, VDI. We populate the parameters of our proposed model with real traces. Using our model, we demonstrate an example of how data accesses vary with time on different virtual disks for different types of virtual desktops. We further validate the usefulness of our model and show we can identify more accurate and fine-grained storage requirements of VDI than current industry methods. Based on the storage requirements identified and the bottlenecks determined, we are able to better guide administrators in their configuration of a storage system to meet the storage requirements with minimum resources.

Chapter 3

Improving Storage Services of Docker Containers and Kubernetes

3.1 Introduction

The use of Linux containers [3, 4] has boomed as many in industry transition from traditional virtual machines (VMs) to this light-weight virtualization technology using containers. For example, Google claims it runs all its software in containers [98]. Linux containers are a virtualization method for running multiple applications in isolated systems (i.e., containers) on a host using a single Linux kernel. Linux containers were initially released in 2008, but their use soared after Docker's release in 2013 [61, 60]. Docker is a platform for creating, deploying, and running applications inside containers. In order to deploy and manage applications in Docker containers across multiple hosts, people use a container orchestrator to perform cluster management and orchestration (i.e., selecting which cluster nodes will host which containers). Kubernetes (k8s) is one prevalent container orchestrator. In k8s, a pod is the basic management unit. It includes a container, or a group of tightly coupled containers, with shared storage and network resources and a specification for how its containers should run.

Containers were initially designed for stateless applications. Due to the success and

popularity of containers and Kubernetes, more and more stateful applications are moving to containers. When deploying stateful applications, users usually have service-level objectives (SLOs) on storage as part of their service-level agreement (SLA) requirements. In the current k8s, users can specify their requirements on CPU, memory, affinities to nodes in the cluster or other pods, etc., in the pod configuration file when deploying a pod. If users want to specify storage requirements, they can refer to a StorageClass(SC) which they think can meet their storage requirements. SC is used to describe the class of a storage. For example, an administrator may classify the storage resources into three classes: gold, silver and bronze. StorageClasses must be pre-created by administrators in the cluster.

The current storage management in k8s has some limitations. First, storage selection is excluded from the host selection process, which does not consider the storage resources that each host can access. During the scheduling process, k8s just assumes the selected host will have access to enough storage resources, which may not be the case. Hosts in the k8s cluster may only have access to a limited number of local storage or shared storage. The accessibility to different storage resources, and the available storage resources should be considered during the pod scheduling process. Second, SC is static and cannot be used to efficiently schedule storage resources, since the actual performance of the storage changes with the utilization of the storage resources. Third, it is hard to decide a proper number of SCs that just satisfies users' requirements without wasting resources. If the number of SCs is small, people have few options and may have to pick SCs that provide more resources than they want. As the number of SCs in a system increases, the storage utilization efficiency might improve due to more user options, but it requires more effort to maintain. Fourth, modern applications may have advanced storage requirements such as rate limiting and caching policies [99, 100]. However, SC does not support these advanced storage requirements. In addition, from the users' point view, it takes extra effort to select a correct SC. More importantly, the SC selection process is error-prone, and it may cause storage SLO violations or wasted storage resources.

Storage resource management has been a research problem in VM environment for years. For example, Pesto [101] provides storage management for VMs running on VMware's vSphere [96]. But the advancement in VM storage resource management

cannot be easily applied to containers. First, when placing a VM disk into storage, VMware uses a profile to describe the storage requirements of a VM. Then, a proper storage compatible with the VM profile will be selected. This process is similar to allocating storage by using SC in k8s. Second, VM storage management systems like Pesto are built upon Hypervisor. In the container environment, orchestration platforms (e.g., Kubernetes, Mesos, and Docker swarm) perform the role of Hypervisor. Applying a VM storage management system to k8s must follow the interface of k8s, and work as a storage provisioner, which relies on SC to describe the storage allocated to users. Therefore, it will share the same limitations of SC. Third, in Pesto, each host is connected to the centrally managed storage. A VM can always access to the allocated storage resources, regardless which host it is running on. Therefore, Pesto does not select storage in the process of scheduling VMs, and thus cannot solve the issue of missing storage selection in the host selection process of k8s.

In order to overcome these limitations in the current storage management of k8s, we propose k8sES (k8s Enhanced Storage), a system that can efficiently support applications with various storage SLOs along with all other requirements deployed in the Kubernetes environment. K8sES allows users to put their detailed storage requirements, including capacity, bandwidth, sharability, advanced policies, etc., directly in their configuration files. As a result, no modifications are made to the universal interface (*kubectl create -f* $\langle manifest \rangle$), which is used to create pods. We redesign the current scheduling and storage allocation mechanisms in k8s, so that it can dynamically allocate storage to applications based on users' storage requirements. At initial storage allocation, k8sES will select appropriate hosts and storage and automatically carve out storage resources to support the running of pods, based on users' requirements and the performance of each host and storage at real time. During the runtime of applications, k8sES can monitor the performance of both pods and storage, and adjust the storage resource allocation based on the storage SLOs compliance.

In summary, our key contributions are:

- K8sES overcomes the limitations in storage management of k8s by providing dynamic storage provisioning to k8s for the purpose of meeting users' storage SLOs.
- K8sES improves the storage utilization efficiency in k8s.

- K8sES saves effort for both k8s administrators and users by improving the complex and error-prone storage allocation mechanisms in the current k8s.
- K8sES provides new monitoring capabilities to monitor the I/O performance of both pods and storage devices in k8s.

The outline of this chapter is as follows: Section 3.2 introduces the background of k8s, and analyzes the current storage support in k8s. We then discuss the scope and objectives in this section. In Section 3.3, we illustrate the design of k8sES. The implementation of our proposed system is described in Section 3.4. We show the benefits of k8sES in meeting users' storage SLOs brought to k8s in Section 3.5. Section 3.6 summarizes the related work. Section 3.7 concludes the paper.

3.2 Background and Motivation

3.2.1 Kubernetes Components

A Kubernetes cluster is composed of master components and node components. Master components provide a control plane to the cluster including a front-end for the control plane (kube-apiserver), a backing store for all cluster data (etcd), a scheduler that schedules pods (kube-scheduler), and a controller manager that runs controllers to watch and ensure the cluster state (kube-controller-manager). Typically, master components run on the same machine. Node components run on all nodes in the cluster. On each node, there is a daemon responsible for creating pods (kubelet), a proxy that forwards TCP or UDP traffic to the pods (kube-proxy), and a container runtime (e.g., Docker). The container runtime provides basic container management functions to k8s, e.g., creating, starting, and stopping containers, managing container images, etc.

3.2.2 Storage Support in k8s

Data in containers are ephemeral. To allow persistent user data, Kubernetes provides a volume abstraction to store data permanently. A volume in k8s outlives any containers that run within the pod, and data is preserved across container restarts. Essentially, a k8s volume is a directory on the host and is mounted into containers of a pod. The medium that backs a volume is determined by the particular volume type being used,

e.g., local, iSCSI, awsElasticBlockStore (Amazon Web Services EBS), gcePersistentDisk (Google Compute Engine Persistent Disk), etc.

K8s provides a PersistentVolume (PV) subsystem to manage how backend storage is provisioned and consumed. A PV is a portion of the clusters storage that an administrator has provisioned [102]. A PV will generally have a specific storage capacity set when it is created. Currently, storage size is the only PV resource attribute that can be set or requested. To avoid exposing users to the details of how volumes are implemented, an administrator usually uses StorageClass (SC) to categorize PVs, e.g., gold, silver, and bronze classes. A PV can be manually provisioned or dynamically provisioned. To manually provision a PV, cluster administrators have to make calls to their storage provider to create storage volumes in advance and then create PV objects to represent them in k8s. In the manifest (i.e., configuration file) of a PV, the administrator will assign an SC name to indicate the StorageClass of that PV. To dynamically provision a PV, cluster administrators first have to create StorageClass objects in k8s. Inside the manifest of each SC object, the administrators must then specify a set of properties of the volume and a provisioner that is able to provision such volumes. Each SC must be unique in the cluster. Regardless how PVs are created, administrators must classify the storage resources in the cluster by using SC in advance.

With PVs, a k8s user does not use the pod manifest to specify what storage should back the volume. Instead, k8s users provision volumes using a PersistentVolumeClaim (PVC). Storage size is the only attribute that can be used in a PVC request, but a user can choose the class of storage by specifying an SC name. After a PVC is created, k8s will look for a matching pre-provisioned PV, or create one if dynamically provisioning PVs, and bind the PVC with it.

When scheduling a pod, k8s only selects a host for the pod based on CPU, memory, affinities requirements, etc. It does not consider the storage resources that each host can access. After a destination host is determined, the kubelet daemon on the selected host will assign a pre-created PV to the pod, or call the corresponding storage provisioner to create one. It assumes that each host has access to enough storage resources.

First, storage selection is excluded from the host selection process, which does not consider the storage resources that each host can access. During the scheduling process, k8s just assumes the selected host will have access to enough storage resources, which

may not be the case. Hosts in the k8s cluster may only have access to a limited number of local storage or shared storage resources. The accessibility to different storage resources, and the available storage resources should be considered during the pod scheduling process.

Given these concepts, the process of deploying an application with persistent storage in the current Kubernetes environment is described as follows: (1) The administrator classify storage resources in the cluster by using SC. (2) The administrator creates PV objects (in manual provisioning) or SC objects (in dynamic provisioning). (3) The user creates a PVC. (4) The user creates one or multiple pods and refers to the previously created PVC. (5) The kube-scheduler selects hosts. (6) The kubelet daemons on the selected hosts assign storage.

3.2.3 Limitations of Storage Support in k8s

With support for storage as it currently exists in k8s, administrators can configure their cluster's storage into different categories and present them to users as multiple StorageClasses. However, this mechanism suffer from several limitations. First, storage selection is not considered when scheduling pods in kube-scheduler. In case that hosts only have access to a limited number of local storage or shared storage, it is probable that the storage resources a selected host can access cannot meet the user's storage requirements. Second, SC is static, but the performance of a storage system dynamically changes as the workload running on it constantly evolves. For example, consider a system containing multiple HDDs and SSDs where the administrator configures two SCs. One SC is called "fast" with SSDs as the backend, and the other SC is called "slow" with HDDs as the backend. If too many workloads are running in the SSDs and make them congested while the HDDs have no workload, the "fast" SC can be slower than the "slow" SC. To avoid this problem, administrators have to monitor the performance of all types of storage and dynamically adjust the physical configuration under each SC. This monitoring and adjustment takes significant effort from the administrators to meet users' storage SLOs. Second, limited SC choices or inappropriate configurations may waste storage resources to meet users' SLOs while large numbers of SCs are hard to maintain. When deploying applications, users have to map their storage requirements into an existing SC, which usually provides more resources than

they need, e.g., more bandwidth, more space, or unnecessary functions (encryption, deduplication, etc.). Otherwise, users will be in danger of an SLO violation. If the number of SCs are few in a cluster, it has a high chance that a user may have to pick an SC providing resources much more than what he/she actually needs. If administrators want to perfectly meet each user's storage requirements, they may have to create an SC for each user with different storage requirements. In this case, it requires a large number of SCs which is hard to maintain. Third, StorageClass does not support advanced storage requirements. Some modern applications have advanced storage requirements expressed as policies [99, 100]. These policies can be dynamic, where an action is triggered if conditions are met. For example, a policy that requires cache when the GET operations per second are greater than 5 can be expressed as:

$$\text{policy: WHEN GETS/s} > 5, \text{SET CACHING} \tag{2.1}$$

With these advanced storage requirements, it is hard to pick an appropriate SC.

3.2.4 Scope and Objectives

This chapter focuses on guaranteeing users' storage requirements in Kubernetes environment. Users' applications are deployed in containers in k8s. We assume the total CPU, memory, and storage resources in the k8s cluster are fixed, and administrators will not add infinite resources due to the high cost and maintenance effort.

K8sES is designed based on several objectives to ensure users' storage requirements are met. First, the storage where a pod is running should meet the storage SLOs of the user. At the same time, the node where the pod is running should meet other k8s resource related requirements of the user, including CPU, memory requirements, etc. Second, all k8s non-resource requirements, including port, affinities with pods, etc., should be met. Third, the pod scheduling and storage allocation decisions should reduce the possibility of SLO violations. Fourth, resources in the cluster should be used efficiently. Fifth, it should be easy for administrators to configure. Sixth, the proposed design of k8sES should keep the same interface as standard k8s. To these ends, k8sES can accommodate different storage SLOs.

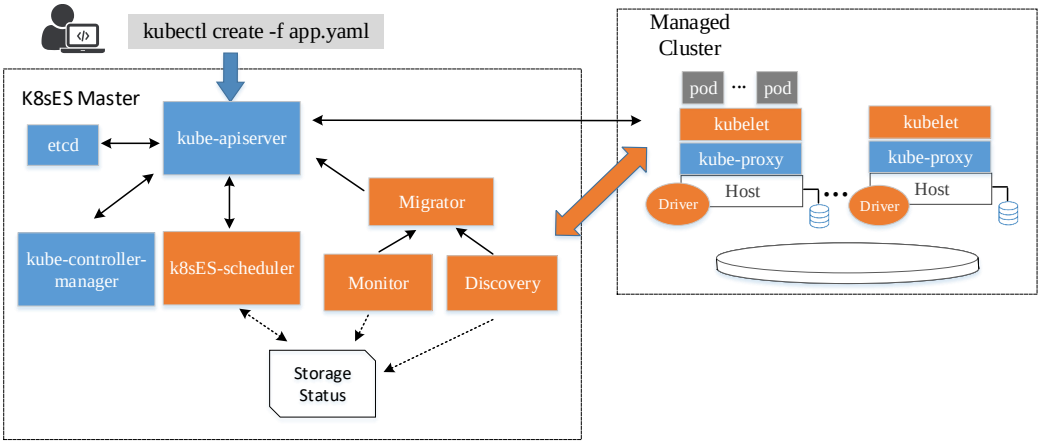


Figure 3.1: System architecture of k8sES.

In this chapter, we focus on several essential storage requirements. K8sES enables users to specify capacity, sustained bandwidth, storage sharability, and advanced storage policies [100] as stated directly in their pod configuration file.

3.3 Architecture

Figure 3.1 shows the architecture of k8sES, which follows the master-slave mode of Kubernetes. In Kubernetes, a user puts the specifications of an object that he/she wants to create in a configuration file (manifest) and calls a universal interface *create -f* $\langle manifest \rangle$ to create the object. Since this interface is easy and succinct, there is no need of modifying the interface; rather, we add a new section in the manifest that allows users to directly specify their detailed storage requirements (e.g., Figure 3.2). When users of k8sES are not certain about which StorageClass to pick, they can simply put their detailed storage requirements in the manifest and use the same standard interface to deploy an application.

After receiving a scheduling request, the k8sES-scheduler (a modified kube-scheduler) is responsible for selecting a proper host that can meet the user's standard computation, memory, and non-resource requirements (affinity, node health, port, etc.) as well as a proper storage device (or system) that can meet the user's storage requirements. The k8sES-scheduler will restrict that the selected host has access to

```
<k8sES volume>
  size: x GB
  sustained bw: y MB/s
  sharing: False
  reclaim: Retain
  policy: WHEN GETS/s > z, SET CACHING
```

Figure 3.2: Example storage requirements in k8sES.

the selected storage. The scheduling decision is sent to the kubelet on the selected host to launch the pods. We extend the kubelet function so it can automatically allocate storage resources for the pods on the selected storage device. After receiving the scheduling decisions, the kubelet will pass parameters about the users' storage requirements to our k8sES driver so it can create volumes with the requested resources on the selected storage. Finally, the kubelet calls the underlying container runtime to launch containers and mount the storage to these containers. The Monitor module monitors all the running pods and storage. It collects I/O related data from each running pod and each storage device. The collected data are used for both pod and storage management. If there is any storage SLO violation, the Monitor module will call the Migrator to migrate pods and data. When deleting a pod, its allocated storage will be retained if users have set the reclaim policy to "Retain" in the new storage section of the pod configuration file. Users can reuse a volume by referring to the name of the pod that this volume previously belongs to, plus the name of the volume itself. If the reclaim policy is "Delete" or omitted, the allocated storage will be deleted together with the pod.

K8sES also contains a Discovery module to detect the available storage resources in the cluster. The joining and leaving of storage devices, as well as storage failures, can be automatically detected by the Discovery module. Both the Discovery and Monitor modules track the remaining storage resources in the system. The k8sES-scheduler queries these two modules to determine the currently available resources on each storage device.

K8sES has high scalability. To avoid single-node limitations, both the k8sES master and worker nodes can be horizontally scaled in the same way as standard k8s [103, 104].

3.3.1 Selecting Storage Along with Nodes

The current k8s process of scheduling a pod on a node undergoes ***predicate***, ***priority***, and ***select*** steps. In the *predicate* step, the scheduler filters out nodes that cannot meet all the predefined predicates (i.e., quick yes/no rules). These predicates will check whether those requirements with definite numbers and those explicitly specified as "RequiredDuringScheduling" in the pod configuration files are met. In the *priority* step, the scheduler checks the priority of each node by scoring it based on a list of priority rules. Each priority rule has a weight and calculates a score from 0 to 10 for each node. The weighted summation of the scores from all priority rules is the final score of a node. Finally, the scheduler selects the node with the highest score and sends the decision to kube-apiserver. The kubelet on the selected node will launch the pod. If there are multiple nodes with the same highest score, the scheduler will select one in a round robin fashion.

In k8sES-scheduler, we select both storage and nodes for a pod. As storage and nodes are different resources, how to coordinate the scheduling to meet users' various requirements is an issue. Intuitively, we may select nodes first. In this case, however, the storage belonging to the selected nodes may not meet the storage requirements. Alternatively, we may select storage first. In this case, the selected storage may not belong to any eligible nodes. In k8sES, we first define users' storage SLOs (capacity, bandwidth, sharing, policies, etc.) as predicates. All other requirements used to reduce the possibility of SLO violations and optimize cloud resource efficiency are defined as storage priority rules (discussed in Sec. 3.3.2). In the *predicate* step, the scheduler will first filter a list of node candidates which can meet all node related requirements. Then, we check the storage predicates. We only check the storage that can be accessed by the filtered nodes. In the *priority* step, we calculate scores for each filtered node and storage device based on the node priority rules and storage priority rules. In the final *select* step, one option to select node and storage is selecting a node with the highest score calculated by the node priority rules and then picking storage with the highest score accessible by this node. However, this method downgrades the importance of storage priority rules. It is possible that all the storage devices that the selected node can access have very low scores (e.g., the available storage space and bandwidth resources are exactly the same as the requested resources). K8sES selection considers both node

and storage priority rules fairly. After calculating the score of a node and the scores of its accessible storage, we pick the storage with the highest score and add its score to the score of the node. The final decision selects the node that has the highest combined score and then selects the storage with the highest score on that node. The decision tuple $\langle Node, Storage \rangle$ is then sent to the kubelet on the selected node where it will launch the pod on $Node$ and create the volume on $Storage$.

3.3.2 Priority Rule

In k8sES, the *predicate* process ensures the storage SLOs can be met at initial allocation. However, as more applications are launched, the storage SLOs of existing pods may still be violated due to bad allocation from other sources. K8sES considers this possibility and gives preference to storage that has a lower possibility of future SLO violations. K8sES sets a *least_storage_usage* priority rule in the *priority* process that assigns a higher score to a storage device if its space and bandwidth usage would still be low after assigning the pod to it. In practice, the score (maximum is 10) of a storage device for this priority rule can be calculated as:

$$\begin{aligned} Score = & (10 \times \frac{Size_{total} - Size_{requested}}{Size_{total}} \\ & + 10 \times \frac{Bandwidth_{total} - Bandwidth_{requested}}{Bandwidth_{total}})/2 \end{aligned}$$

where $Size_{total}$ is the total size of a storage device, $Size_{requested}$ is the requested storage size of all existing pods using that device plus the pod to be launched, $Bandwidth_{total}$ is the total bandwidth of the storage device, and $Bandwidth_{requested}$ is the requested storage bandwidth of all existing pods using that device plus the pod to be launched.

Moreover, storage and other resources in the cluster need to be balanced. Otherwise, an unbalanced utilization may waste resources. For example, assume there is a cluster of two nodes that each have local storage. Each node has 64GB of memory and 1TB of storage, and we only care about memory and storage space resources. Due to system and user activities, the current available resources are 38GB of memory and 900GB of storage in Node 1 and 58GB of memory and 400GB of storage in Node 2. Now

Table 3.1: Example scores of nodes with different resources

Nodes	Memory	Storage Space	Combined
Node 1	1	8	9
Node 2	4	3	7

assume a pod requires 32GB of memory and 100GB of storage. The scores given by the *least_storage_usage* rule on memory and storage are summarized in Table 3.1. As discussed in the previous section, we select Node 1 as it has a higher combined score. This pod will be launched on Node 1 and its local storage. After this allocation, only 6GB of memory is left while 800GB of storage is unused on Node 1. If all incoming pods require more than 6GB of memory, the remaining storage resources on Node 1 will be wasted. Therefore, we set a *usage_leveling* priority rule which assigns a higher score to a storage device if CPU, memory, storage space, and storage bandwidth usages will be more balanced after assigning the pod to it. In practice, the score of a storage device under this priority rule can be calculated as:

$$Score = 10 - 10 \times |\frac{CPU_{requested}}{CPU_{total}} + \frac{Memory_{requested}}{Memory_{total}} - \frac{Size_{requested}}{Size_{total}} - \frac{Bandwidth_{requested}}{Bandwidth_{total}}|$$

Each priority rule is also associated with a weight (0 to 1). Administrators of k8sES can adjust the importance of each priority rule by adjusting the weights. When adding the storage score to the score of a node, we also have weights allowing administrators to adjust the importance of storage resources and non-storage resources. By setting these weights during the startup of k8sES, people can achieve different trade-offs between storage resources, node resources, resource usage efficiency, risk of SLO violations, etc.

3.3.3 Discovery

Unlike k8s, k8sES does not require administrators to create PersistentVolume or StorageClass resources to describe and categorize the storage capabilities in the cluster. We

include a Discovery module to detect all available storage resources in the cluster automatically. First, when a node joins the cluster, it must register with the Discovery module to report its available storage resources including Name, Location, Size, Bandwidth, and Shareability. Whenever storage is added to or removed from a registered node, the node also needs to report the changes to the Discovery module. Expanding on the node health check capabilities in the current k8s, the Discovery module allows k8sES to also detect storage failures. Each node periodically sends heartbeat messages to the Discovery module. The module maintains a liveness map of each storage device and is quickly able to recognize storage failures. If there is a node or storage failure, the Discovery module will call the Migrator module to start the failover process.

3.3.4 Monitoring

The Monitor module in k8sES collects the I/O performance of both pods and storage devices. It collects the read and write I/O throughput of each pod in real time. It also monitors the collective I/O throughput and space utilization of each storage device. The collected data are used in four ways.

First, the collected data are used to identify any misbehaving pods. In case that a pod takes more I/O resources than it requested, the Monitor may throttle the I/Os, or it may log and report the event to the administrator for later auditing. The actions to be taken are determined during the initialization of k8sES.

Second, the collected data are used to enforce dynamic policies when their conditions are met. The Monitor module analyzes the I/O performance related statistics, e.g., read/write IOPS (I/Os per second) and read/write throughput. Once the metric defined in the dynamic policy of a pod meets the condition, the Monitor will call the corresponding functions or tools to take actions. For example, assume a pod defines a dynamic policy (2.1). Once the read IOPS on the storage allocated to that pod is greater than 5 for a significant amount of time (e.g., one minute), the Monitor will set up a dedicated cache for the pod. Note that such a cache is more useful for pods accessing remotely shared storage. For pods accessing local storage, such a policy may be ignored due to the existence of the file system cache. Currently, the functions in k8sES include caching and I/O throttling. In the future, we may support more metrics by collecting richer information in the cluster and support additional I/O related functions.

Third, the collected data are used to adjust the storage resource allocation based on the actual storage resource utilization. These adjustments are discussed in Sec. 3.3.5.

Fourth, the collected data are used to determine possible storage SLO violations of pods. In case of SLO violation, it will call the Migrator module to do migration. The details are discussed in Sec. 3.3.6.

3.3.5 Thin Provisioning, and Multiplexing

The Monitor module analyzes storage utilization related statistics, e.g., disk space used by each pod and read/write throughput of the storage on each host. It maintains two tables: one about the space usage at the pod level and one with bandwidth usage at the storage level. For each pod i running in the cluster, it records the current disk space used on the file system, S^i_{used}, and the requested storage space, S^i_{req}. After the scheduling, the kubelet on the selected node will not fully provision the volume with the requested space. Instead, it will only provision a volume with size $\rho \cdot S^i_{\text{req}}$ where $(0 < \rho \leq 1)$. ρ controls the initial storage space allocation. As time goes on, once the file system usage reaches a threshold θ, the Monitor module will issue a command to the host to expand the current storage space allocation by $\mu \cdot S^i_{\text{req}}$ where $(0 < \mu \leq 1)$. μ controls the speed of space allocation increase. This method of storage space allocation is called "thin provisioning." Thin provisioning is reasonable because users typically request more storage space than they actually use. Thus, thin provisioning can further save storage space resources of the cluster. There are still tradeoffs, however. Considering that increasing storage space of a volume while it is being used may influence the ongoing I/Os, a big ρ and μ will ensure more steady I/O performance but waste more storage space. A small ρ and μ can save more storage space but may hurt the I/O performance. The configurations of ρ and μ are at administrators' discretion and can be adjusted in k8sES.

For each storage device j detected by the Discovery module, the Monitor module records its throughput (TP in MB/s) at time t as TP^j_t, its total requested bandwidth as B^j_{req}, and its literal (overall maximum) total bandwidth as B^j_{total}. The Monitor module calculates the average throughput, $\overline{TP^j}$, over a time interval τ (e.g., six hours) from the TP^j_t measurements for each storage device. With these parameters, the Monitor module calculates the bandwidth utilization as:

$$\frac{1}{\alpha^j} = \frac{\overline{TP^j}}{B^j_{\text{req}}}$$

We call α^j the amplification factor of the bandwidth of storage device j. It indicates that we can allocate pods to storage device j as if it has a total bandwidth of $B^j_{\text{amlified}} = \alpha^j \cdot B^j_{\text{total}}$ without violating pods' storage bandwidth requirements. We use B^j_{amlified} to update the available bandwidth for storage device j, on which the scheduling process is based. Conceptually similar to thin provisioning, we call this scheme of storage bandwidth allocation "multiplexing." Given that users typically request more storage bandwidth than they actually use, this form of statistical multiplexing will further save storage bandwidth resources of the cluster.

Since α^j keeps changing as $\overline{TP^j}$ changes, we will calculate a new B^j_{amlified} only when the difference between the newly calculated amplification factor α^j and the one currently being used to calculate B^j_{amlified} is greater than a threshold γ (e.g., $\pm 10\%$). A large γ will lead to late updates of B^j_{amlified} and further cause either resource waste (due to decreased bandwidth utilization) or SLO violations (due to increased bandwidth utilization). A small γ will lead to frequent updates of B^j_{amlified} and a waste of computation power. In practice, we set γ to be 10%. In addition, it is possible that at some earlier time the bandwidth utilization of a storage device is very low, and a lot of pods are allocated to this storage. This results in a high ratio between B^j_{req} and B^j_{total}. Once a pod's throughput resumes to its requested bandwidth, there is a high chance that a lot of pods' storage SLOs will be violated. To reduce the chance of such SLO violations, we set a cap for α^j to be no more than 120%.

3.3.6 Migrator

Both thin provisioning and multiplexing improve the storage utilization efficiency but bring potential storage space or bandwidth SLO violations. To meet users' storage SLOs with high storage utilization efficiency, we design a Migrator to migrate pods along with storage. If the total space utilization of a storage device reaches a threshold (e.g. 90%), it will trigger the migration. This is because there is a chance that the next storage space expansion of a pod may fail due to insufficient storage space. The migration will

also be triggered if TP_t^j equals B_{total}^j, which means the current stable throughput on storage device j has reached the literal maximum. In other cases like node and storage failure, pods and storage will also be migrated.

If the migration is triggered by either thin provisioning or multiplexing, the Migrator has to migrate the storage of one or more pods. When selecting candidates to migrate, several factors need to be considered. First, migrating the storage of a pod will freeze the pod's I/O for a period. The bigger the storage space allocated, the longer it takes to migrate. Second, migrating a pod with bigger storage will release more storage space. Third, migrating a pod with a higher throughput will release more bandwidth resources. Fourth, some pods have pod affinities that require them to stay on the same node. Our goal of migration is to reduce the down time of a pod and reduce future migration. Our migration candidate selection algorithm works as follows.

(1) If the migration is triggered by thin provisioning, migrate the pod(s) with the smallest storage space allocated. No matter which pod is migrated, some storage space will be released. Because migration is done early at the 90% threshold, no storage space SLO has been violated yet and the migration scheme can conservatively release storage space to reduce migration overhead. For a pod with pod affinity, we consider all those affinities as a group. We sum the allocated storage space of that group and compare it with other candidates. If the group is selected to migrate, we migrate the whole group.

(2) If the migration is triggered by multiplexing, we sort pods based on space allocated and current throughput, respectively. We assign scores based on the rank. A smaller allocated storage space has a higher score, and a higher throughput has a higher score. Then, these two scores are summed and the pod with the highest sum will be migrated. For a pod which has pod affinity, we will treat all those affinities as a group.

The migration destination is determined by the kube-scheduler as a new scheduling process except that the originally selected storage is excluded.

3.4 Implementation

We implement k8sES based on Kubernetes Release-1.7. The current k8s implementation is decoupled into multiple binaries. Each module of k8s is an individual binary as are the new/enhanced k8sES modules.

Our implementation of the Discovery module and Monitor module in k8sES uses master/slave mode. Each k8sES node runs a Discovery slave and a Monitor slave daemon. The slaves collect the performance measurements for each pod and the storage information on each node, and they send the data to the Discovery master and Monitor master. If the masters make some decisions, they call the corresponding module to carry out the operations. In the current implementation, the Discovery slaves run *lsblk* to get the storage capabilities. The Monitor slaves run *iotop* on each node to monitor the I/O performance of each pod and run *dstat* to monitor the I/O performance of each storage device. People may extend the functions of k8sES by calling other tools. The Discovery master and Monitor master run in the same node as the k8sES master. They maintain a data structure recording the storage configuration map of all nodes in the cluster. This data structure is shared with the k8sES-scheduler module. In addition, the Monitor master also maintains a map of all running pods containing the current I/O performance and the dynamic policies of each pod. It will call the corresponding slaves to execute the actions defined in the policies if the conditions are met. Since the Monitor master is an individual binary, people can easily extend the support of storage policies without affecting other functionalities of k8sES. The Discovery master will call the Migrator module if any node or storage is inaccessible. The Monitor master will call the Migrator module if there are storage SLO violations.

The k8sES-scheduler receives the storage configuration map from the Discovery and Monitor modules. It maintains the map by recording the currently available resources of each storage device. We implement our k8sESdriver as a Flexvolume plugin. Flexvolume lets users write their own drivers to make Kubernetes support their volumes [105]. Once called by the kubelet, the k8sESdriver will first call the interface of the selected storage device or system to create volumes with the required size. For example, if LVM is the backend storage and storage groups are presented, the volume driver will create logical volumes to be used by the pod. It then optionally creates a file system (if required in the pod configuration file) and mounts the volume as a k8s volume. If there are sustained bandwidth requirements that need to be met, it will also set cgroups [106] to limit the bandwidth of the pod accessing this volume. After the volume driver mounts the created volume to the mount point defined in the pod configuration file, the kubelet then starts the containers defined in the pod.

3.5 Prototype Evaluation and Comparison

This section evaluates a prototype of k8sES. Section 3.5.2 validates the effectiveness of k8sES in meeting users' storage SLOs. We compare the scheduling results of k8sES with standard k8s. Section 3.5.3 shows the effectiveness of I/O throttling in k8sES and its benefits when sharing storage. We show the I/O monitoring abilities, storage resource savings, and migration capabilities of k8sES in Section 3.5.4. Section 3.5.5 tests the storage usage efficiency under different circumstances. We discuss the overhead of k8sES compared with k8s in Section 3.5.6.

3.5.1 Experiment Setup

Table 3.2: Storage configuration of k8s cluster

k8s workers	CPU	MEM	Local Size	Local Bandwidth	Shared Storage
Worker 1	2	2GB	70GB	20MB/s	Share same 100GB at 50MB/s
Worker 2	2	2GB	50GB	50MB/s	
Worker 3	2	2GB	20GB	100MB/s	
Worker 4	6	4GB	50GB	50MB/s	Share same 100GB at 50MB/s
Worker 5	3	3GB	70GB	20MB/s	
Worker 6	3	2GB	50GB	50MB/s	
Worker 7	2	2GB	10GB	100MB/s	

Our testbed is seven virtual machines (VMs) running on two physical servers. Each server has two six-core Intel Xeon 2.40GHz E5-2620 v3 CPUs, 64GB of memory, 1TB Seagate ST1000NM0033-9ZM173 SATA hard disk, and is connected to an HP ProCurve 5406zl switch through a 1Gb/s Broadcom NetXtreme BCM5720 NIC port. Among these VMs, six run as k8s workers, and one runs as both the k8s master and a worker. All physical servers and VMs run Ubuntu 18.04. These seven VMs form a k8s cluster. For each worker node, we allocate a local and a shared storage space with different capabilities. Each workers total available CPU, memory, and storage resources are summarized in Table 3.2.

3.5.2 Validation

We first verify scenarios where k8sES is needed to schedule pods to the correct node and storage in order to meet the pods' various requirements. To simplify analysis, we schedule pods on Workers 5 and 6.

In the first scenario, we focus on the storage capacity requirements. We sequentially deploy pods A1, A2, A3, and B. Pods A1, A2, and A3 each require 5GB of non-shared storage and 2MB/s of storage bandwidth ($\langle 5GB, 2MB/s, \text{non-sharing} \rangle$). Assume we have already deployed A1, A2, and A3. Now we are going to deploy Pod B, which requires $\langle 50GB, 10MB/s, \text{non-sharing} \rangle$ storage. The scheduling results of these four pods in k8s and k8sES are listed in Table 3.3a and 3.3b, respectively. In the tables, the remaining storage capacity and remaining storage bandwidth of the workers are given before Pod B is deployed. The current k8s scheduler does not consider storage resources. Both Worker 5 and Worker 6 will pass the predicate step and have equal priority scores in the priority step when deploying these four pods. Thus, k8s schedules these pods in a round robin fashion. Based on the scheduling sequence, k8s will schedule Pod B on Worker 6. However, Worker 6 only has 45GB of storage capacity available before scheduling B. Such a scheduling decision will violate the storage capacity requirement of Pod B. In contrast, k8sES-scheduler considers the storage resources. Although both workers can pass the predicate step of k8sES-scheduler, the priority step favors Worker 6 when scheduling pods A1, A2, and A3 as it has more balanced capacity and bandwidth resources. When deploying Pod B, Worker 6 will not pass the predicate check as it only has 35GB of storage space available. Pod B will be deployed on Worker 5, and the storage requirements can be met.

The second scenario verifies that k8sES can correctly schedule pods with storage bandwidth requirements. We deploy pods C1, C2, and D in sequence, each requiring $\langle 10GB, 20MB/s, \text{non-sharing} \rangle$ storage. Tables 3.4a and 3.4b show the available storage resources before scheduling pod D and the scheduling results in k8s and k8sES, respectively. Similar to the results in Table 3.3, the scheduling decision made by k8sES is able to meet the storage requirements of all pods while k8s causes a violation of the bandwidth requirement of Pod D.

In the third case, we verify that k8sES is able to meet storage requirements along

Table 3.3: Comparing meeting storage capacity requirement

Workers	Capacity (before B)	Bandwidth (before B)	Result
Worker 5	60GB	16MB/s	A1,A3
Worker 6	45GB	48MB/s	A2,B (✗)

(a) k8s scheduling result

Workers	Capacity (before B)	Bandwidth (before B)	Result
Worker 5	70GB	20MB/s	B
Worker 6	35GB	44MB/s	A1,A2,A3

(b) k8sES scheduling result

Table 3.4: Comparing meeting storage bandwidth requirement

Workers	Capacity (before D)	Bandwidth (before D)	Result
Worker 5	60GB	0	C1,D (✗)
Worker 6	40GB	30MB/s	C2

(a) k8s scheduling result

Workers	Capacity (before D)	Bandwidth (before D)	Result
Worker 5	60GB	0	C2
Worker 6	40GB	30MB/s	C1,D

(b) k8sES scheduling result

with other requirements. We deploy pods E1, E2, and F in sequence, each requiring $\langle 10GB, 20MB/s, \text{non-sharing} \rangle$ storage. In addition, each pod also requires one CPU core and 1GB of memory. Tables 3.5a and 3.5b show the available CPU, memory, and storage resources before scheduling Pod F and the scheduling results in k8s and k8sES, respectively. The schedule results in k8s are made solely based on the CPU and memory resources. After deploying Pod E1 and E2, it schedules Pod F on Worker 5 as it has more balanced CPU and memory resources based on the internal *BalancedResourceAllocation* priority rule of k8s. However, Worker 5 has no more allocable storage bandwidth thus violating the storage bandwidth requirement of Pod F. In contrast, k8sES-scheduler considers various requirements and is able to pick Worker

Table 3.5: Comparing meeting CPU+Memory+Storage requirement

Workers	CPU (before F)	Memory (before F)	Remaining Capacity (before F)	Remaining BW (before F)	Result
Worker 5	2	2	60GB	0	E1,F (✗)
Worker 6	2	1	40GB	30MB/s	E2

(a) k8s scheduling result

Workers	CPU (before F)	Memory (before F)	Remaining Capacity (before F)	Remaining BW (before F)	Result
Worker 5	2	2	60GB	0	E1
Worker 6	2	1	40GB	30MB/s	E2,F

(b) k8sES scheduling result

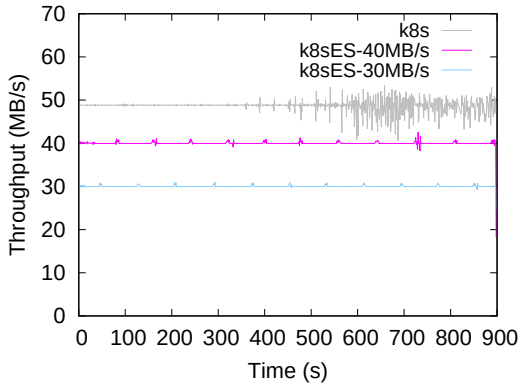


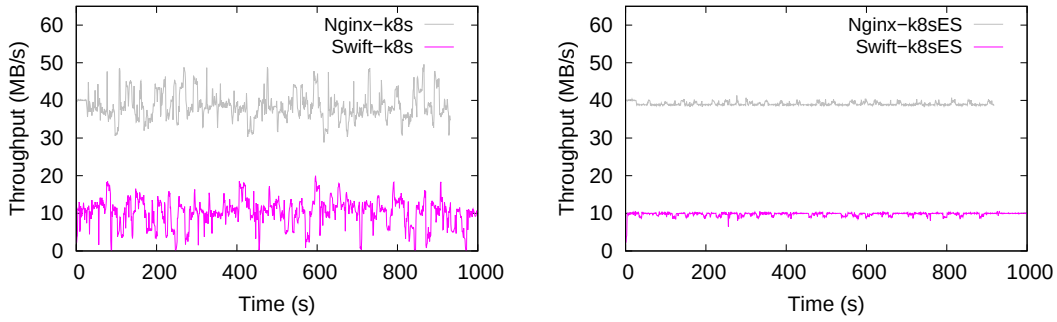
Figure 3.3: I/O throttling in k8s and k8sES.

6, which can meet the CPU, memory, and storage requirements of Pod F.

These are just three possible scenarios where k8sES meets SLOs that k8s cannot. Since users may have storage requirements with any values, SLO violations in k8s may be very common without k8sES.

3.5.3 I/O Throttling

In k8sES, the administrator can configure the Monitor to throttle I/Os or just report the events for future audit when a pod consumes more storage bandwidth than it requested. We first test the effectiveness of throttling I/Os. In k8s, pod I/Os can only be throttled to the disk bandwidth. In comparison, k8sES can throttle I/Os according to users' requirements. Figure 3.3 compares the I/O throttling between k8s and k8sES. We run



(a) k8s (b) k8sES

Figure 3.4: The effects of misbehaved applications on well-behaved applications.

Nginx [107] (version 1.15.9), a popular web server, on Worker 4 with different I/O bandwidth requirements. We set up an HTTP client to download files from the Nginx server. Without any limitation, it will try to download files and generate I/Os on Nginx as fast as possible. When running in k8s, the Nginx pod is set to require local storage and a sustained 40MB/s of storage bandwidth. But the actual I/O throughput of Nginx is bounded by the maximum bandwidth of the local storage on Worker 4, which is 50MB/s. When running in k8sES, we set the I/O bandwidth requirement to be sustained 30MB/s and then 40MB/s for another test. Figure 3.3 shows the actual throughput is throttled to the requested bandwidth. In addition, we can see that the I/O throughput of Nginx in k8sES is more stable than in k8s. With its I/O throttling capabilities, k8sES is able to limit the resources that a particular application can access and thus save resources for other tasks. This is especially important when the resources and budget of the shared cluster are limited.

To show the benefits of I/O throttling when multiple pods share one storage device, we deploy two applications sharing the same storage on Worker 4. One application is OpenStack Swift [108] (version 2.20.0). We use its default benchmark tool, *ssbench* [109], to generate storage requests. The tool, ssbench, performs CREATE, READ, WRITE, DELETE of an object based on a configuration file called scenario. In the scenario, it mostly generates READ requests with 20 workers issuing requests concurrently. We set Swift The other application is Nginx [107] (version 1.15.9). We set up an HTTP client to download files from the Nginx server at a stable rate of 40MB/s. Both applications run in

pods and store their data in the storage space requested from Kubernetes. Swift requests a storage with 10MB/s bandwidth. Nginx requests a storage with 40MB/s bandwidth. In Figure 3.4(a), we can see that the Nginx is often running below the I/O generation speed in k8s, and sometimes can only delivers files at the speed of 3/4 of the requested bandwidth. Its throughput is fluctuating due to unstable performance of Swift. When Swift tries to generates I/Os more than its requested bandwidth (misbehaved), the storage SLO of Nginx is violated. In comparison, with k8sES (Figure 3.4(b)), the I/O throughput of Swift never exceeds its requested bandwidth, and thus the Nginx can always deliver data at a speed that matches the client requirement.

3.5.4 Monitoring, Thin Provisioning, Multiplexing, and Migration

As monitoring capabilities are essential for ensuring storage SLOs and enhancing storage utilization efficiency, we perform an experiment to show these capabilities at both pod and storage granularities. We also show the effects of thin provisioning, multiplexing, and pod migration, which rely on these monitoring capabilities. In this experiment, we run three applications in three pods, namely Pod A, Pod B, and Pod C. Pod A requires $\langle 10GB, 50MB/s, \text{non-sharing} \rangle$ storage. Pod B requires $\langle 2GB, 40MB/s, \text{non-sharing} \rangle$ storage. Pod C requires $\langle 5GB, 30MB/s, \text{non-sharing} \rangle$ storage. Each pod will generate I/Os bounded by its requested bandwidth for 40 minutes. The I/Os follow four normal distributions, each lasting ten minutes. In the first ten minutes, the average throughput of each pod equals half of its requested storage bandwidth. In the third ten minutes, the throughput of each pod will reach its maximum (the requested bandwidth). During the other times (10-20 and 30-40 minutes), the distribution has random mean and standard deviation. Worker 3 and Worker 4 are used to host these applications. On Worker 4, there are already pods fully utilizing 10MB/s of storage bandwidth and 45GB of storage. In the experiment, we deploy applications in the sequence of Pod A, Pod B, and Pod C. These settings will make sure that all these pods will be scheduled on Worker 3 as long as there are enough resources. We set the time interval τ to be five minutes, which is the length of time used to calculate the average I/O throughput, $\overline{TP}$, on a storage device. We deploy Pod B and Pod C after Pod A has run for five minutes.

Figure 3.5 shows the I/O throughput of each application. Figure 3.6 shows the I/O throughput on the local storage of Worker 3 and Worker 4. In the experiment, all three

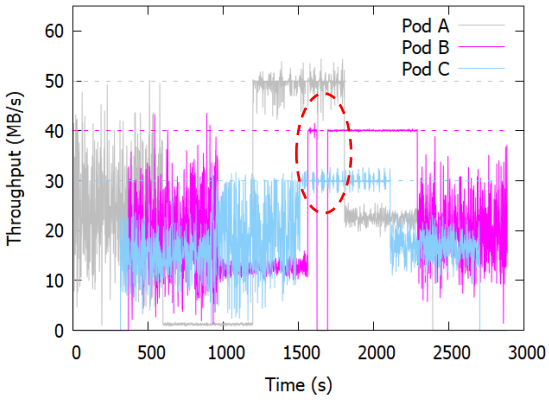
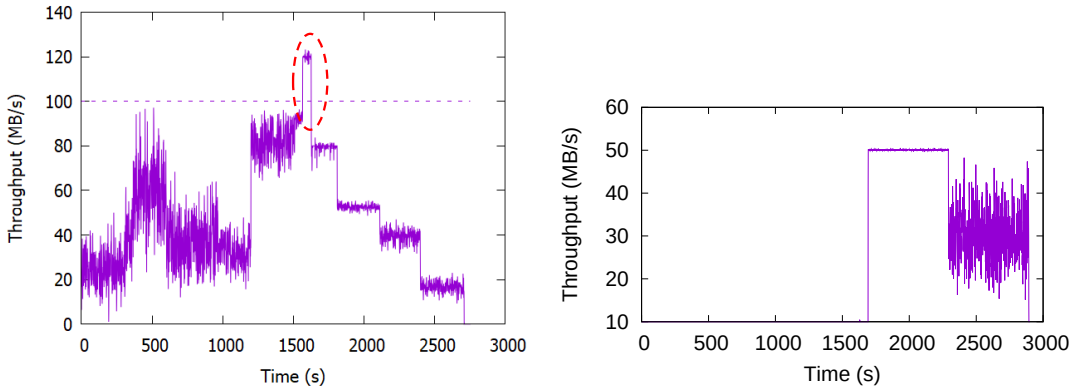


Figure 3.5: Throughput of applications over their lifetime.



(a) Worker 3

(b) Worker 4

Figure 3.6: Monitored I/O throughput on Worker 3 and 4.

applications are scheduled on Worker 3. Note that the sum of the requested bandwidth of the three pods is 120MB/s, which is greater than the 100MB/s literal maximum storage bandwidth of Worker 3. This is because the storage bandwidth utilization of Worker 3 is only half of the requested bandwidth, which results in amplification factor $\alpha = 2$. Due to the 120% cap of α, the thin provisioning and multiplexing mechanism enables allocation of pods to Worker 3 as if it has a storage bandwidth of 120MB/s, which is 120% of its literal bandwidth. We can see that the I/O throughput on Worker 3 never reaches its literal maximum until 20 minutes after the startup of Pod B. At that point, circled in Figure 3.5 and Figure 3.6(a), the I/O throughput of each pod starts

to fully reach its requested bandwidth. When all pods reach their maximums, the I/O throughput on Worker 3 exceeds its literal maximum. This triggers the migration process. Because Pod B has the highest migration score, the Migrator decides to migrate Pod B to Worker 4. The circled region in Figure 3.5 also shows the migration process. The I/Os of Pod B first drop to zero and then resume to its maximum after restarting on Worker 4. After the migration, the I/O throughput on Worker 3 immediately drops below its literal bandwidth, while the I/O throughput on Worker 4 increases due to I/Os from Pod B.

3.5.5 Resource Usage Efficiency

In this experiment, we compare the storage resource utilization of k8sES with the current storage allocation mechanism in k8s. To make a fair comparison, we turn off the thin provisioning and multiplexing functionalities. For the k8s mechanism, which uses SC and PV, we create an SC for each storage device in the cluster and assume the administrator divides the resources of each SC evenly into multiple PVs. We also assume users have full knowledge of the configuration and capabilities of each SC and are always able to make the best decision regarding which SC can meet their storage SLOs with the smallest amount of storage resources.

We deploy four types of applications in the cluster with storage settings as shown in Table 3.2. App1 has two pods requiring $\langle 3GB, 5MB/s\rangle$ storage and $\langle 5GB, 3MB/s\rangle$ storage, respectively. App2 has two pods requiring $\langle 5GB, 10MB/s\rangle$ storage and $\langle 10GB$, $5MB/s\rangle$ storage, respectively. App3 has two pods requiring $\langle 10GB, 20MB/s\rangle$ storage and $\langle 20GB, 10MB/s\rangle$ storage, respectively. App4 has four pods, three of them requiring $\langle 1CPU, 1GB\ Mem\rangle$ and $\langle 1GB, 2MB/s\rangle$ storage, and one requiring $\langle 0.1CPU, 512MB\ Mem\rangle$ and $\langle 15GB, 30MB/s\rangle$ storage. All storage requests are non-sharing.

Figure 3.7 shows the number of instances we can deploy for each of these application types under different configurations. 1 PV and 2 PVs mean we divide each SC into 1 PV or evenly into 2 PVs. "Optimal" shows the maximum number of instances that can be deployed if we evenly divide the SC. For different applications, the number of PVs at "optimal" is different. "Optimal+1" shows the case where we have one more PV than "optimal" under each SC. "K8sES-no-leveling" shows the case where we do not set the *usage_leveling* priority rule in k8sES.

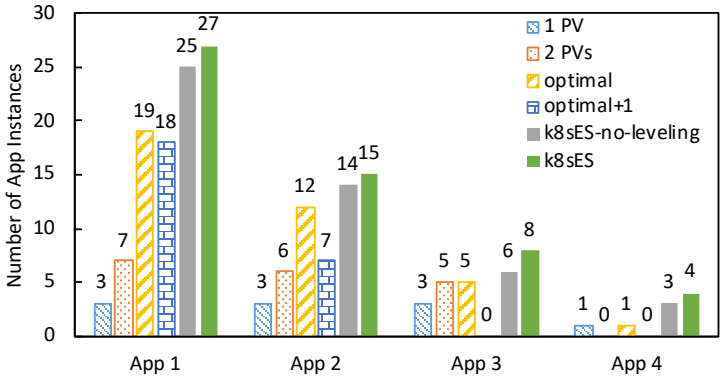


Figure 3.7: Number of applications that can be deployed.

This figure shows that k8sES can deploy the most instances for each type of application. It has a higher utilization efficiency than the optimal case using SCs divided into even-sized PVs. If the SCs and PVs can be created arbitrarily, either automatically or manually, the optimal result is then the same as k8sES without thin provisioning and multiplexing. This is because k8sES allocates storage on the fly based on users' requests. No resources are pre-allocated or pre-created. Furthermore, k8sES may more efficiently utilize available storage with thin provisioning and multiplexing enabled, as described in the previous subsection. Comparing "k8sES" with "k8sES-no-leveling," we see that k8sES has a higher resource utilization efficiency with the *usage_leveling* priority rule enabled. Take App4 as an example. Three pods are CPU and Mem intensive, while one pod is storage intensive. If we do not have the *usage_leveling* priority rule, some nodes (Worker 6 in this experiment) tend to be occupied by CPU and Mem intensive pods but have underutilized storage resources. In the experiment, k8sES is able to place the storage intensive pods in those nodes (e.g., Worker 6), thus resulting in the deployment of one more instance of App4.

3.5.6 Computation Overhead

Fast creation time is a major advantage of containers over VMs. In k8s, containers are created during the pod creation process. In this subsection, we evaluate the influence of selecting storage on pod creation. We create a pod in both k8s and k8sES. We measure the time between calling the "kube create" command and the pod entering "Running"

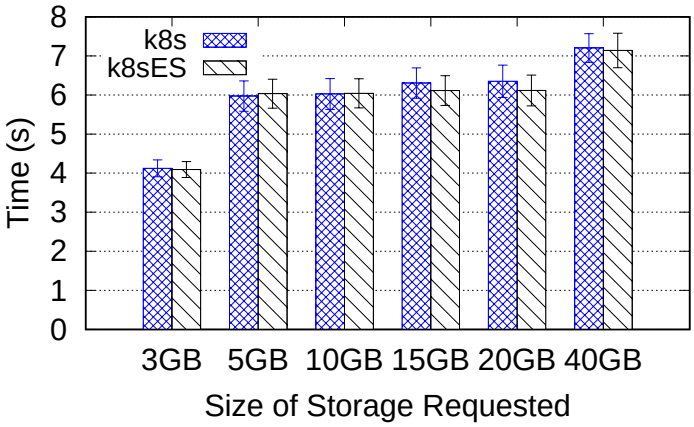


Figure 3.8: Pod creation time with 95% confidence interval.

status. We set the pod to request different sizes of storage. The measurement of the pod creation process is repeated 100 times for each pod configuration in both k8s and k8sES. To make a fair comparison, we install the same driver that we implemented for k8sES in k8s so that volumes will be automatically provisioned in both systems. The difference is that there is no storage selection process in k8s. In k8s, we statically select a storage device for each node that has enough resources for the pod we create. Figure 3.8 shows the average time to create a pod and the 95% confidence interval. We can see the pod creation time in k8sES is similar to that in k8s. Note that k8sES is sometimes a little faster (3.69% at most) and sometimes a little slower (1.03% at most) in creating a pod. This is because our added storage searching functions in k8sES add some minor overhead to the scheduling process. On the other hand, by filtering out nodes that cannot access storage that meets the more demanding user requirements, we may reduce the search space in the *priority* and *select* steps compared with k8s.

3.6 Related Work

Containers offer an efficient way to run applications as microservices. They are the essential building blocks of clustering tools like Kubernetes. A performance analysis by IBM shows that containers perform equal to or better than VMs in CPU, memory, network, and storage related tests [110]. Most container research focuses on Docker

containers. DRR [111] tries to improve the copy-on-write performance in Docker. Research by Tarasov et al. [112] focuses on the choice of Docker storage driver. Other studies like Slacker [113] and Anwar et al. [114] focus on the Docker registry and the image pulling process. Makin et al. [115] study Docker live migration.

Kubernetes is an open-source container orchestration engine developed by Google and evolved from their prior work on Borg [116] and Omega [117]. Since its v1.0 launch in 2015 [118], Kubernetes has become one of the most prevalent container management systems and motivated a handful of academic studies.. Vctor et al. [119] analyze the performance of the Kubernetes system and study its adaptive application scheduling [120]. Xu et al. [121] attempt to manage network bandwidth for Kubernetes. Tsai et al. apply Kubernetes in fog computing platforms for IoT (Internet of Things) [122].

Storage management in Kubernetes is still underexplored. Some third parties provide storage support in Kubernetes. REX-Ray [123] provides a vendor agnostic storage orchestration engine aiming to provide persistent storage for Docker, Kubernetes, and Mesos. Any volume that is to be used by a Kubernetes resource must be previously created and discoverable by REX-Ray. This is similar to the manual provisioning of PVs in k8s. NetApp's Trident [124, 125] provides persistent storage support to k8s. Users can specify Trident as a storage provisioner in StorageClass, so PVs can be dynamically provisioned from supported NetApp storage systems. However, Trident and similar provisioners still suffer from the issues in PV and SC. Our study targets the PV and SC issues in k8s. Trident and similar provisioners can better ensure users' storage SLOs by providing storage backends to k8sES.

In addition, there are storage management systems in VM environment. Pesto [101] is implemented as part of VMware's Storage DRS [126] component of vSphere [96]. It provides an automated storage management system that can model and estimate storage performance, and recommend VM disk placement and migration in order to balance space and I/O resources across the datastores in VMware environment. However, in order to apply the Storage DRS to Kubernetes, a storage provisioner which supports Storage DRS must be developed based on the PV abstraction and SC of Kubernetes. In this way, it still suffers from the limitations of PV and SC we discussed in this chapter.

3.7 Conclusion

This chapter presents k8sES, a system that can efficiently support applications with various storage SLOs along with all other requirements deployed in the Kubernetes environment. With k8sES, users can put their storage requirements directly in their configuration files when deploying applications. K8sES will schedule a pod to the right host and storage, and it automatically creates volumes with the required resources on the selected storage. Users' storage SLOs can be ensured together with all other requirements. In addition, k8sES improves the cluster resource usage efficiency in the cloud and provides I/O monitoring capabilities to Kubernetes.

Chapter 4

Improving Latency SLO with Integrated Control for Networked Storage

4.1 Introduction

In recent years, networked storage has become a popular type of storage. In this environment, storage is connected to clients through network. Cloud storage [6, 63, 64, 65], datacenter SAN and NAS, object storage are typical networked storage. When accessing networked storage, an I/O request will go through the client side I/O stacks, transit through the network, traverse the storage servers and finally be served by storage devices like disks. The response of the request also has to go through these components on the reverse path back to the client. This long I/O path and the diverse components along the path result in increased end-to-end management complexity.

Such increased complexity makes meeting latency SLOs (Service Level Objectives) harder in this networked storage environment. First, it requires all components along the I/O path to react on incoming I/Os in a coordinated fashion. Second, the status of each component is dynamically changing. Third, each component treats I/O requests differently since they have different semantics.

Currently, there are several mechanisms trying to ensure SLOs for network and

storage individually. However, such individual control may not be effective and even bring in more issues. For example, duplicating I/O requests and returning the fastest response from multiple storage replicas may reduce the latency in storage access time but cause congestion on network by creating more network traffic. On the other hand, a solution may be obtained by carefully considering all components along the I/O path with a systematic control that efficiently utilizes available resources at some components while tolerating some performance degradation at other components. For example, we can allow some I/Os to be served with priority in the congested network so that they can be processed by storage devices sooner and returned to the client in time. Studies like [99, 127, 128, 129] consist of control on both network and storage when trying to ensure SLO. However, further study is still needed on how to coordinate network and storage and fully utilize their characteristics. We believe that a systematic way of coordinating all of the involved components is necessary in order to ensure SLO in this environment.

In this chapter, we introduce JoiNS, a system that coordinates different components along the I/O path to meet latency SLO in a networked storage environment. We bridge the gaps between different components by deploying a logically centralized controller to orchestrate the control to each component. The controller has a global view of the network and storage status. It keeps collecting information of the current network, and storage status and estimates the latency at network and storage respectively for each I/O request. It determines whether to control I/Os based on the latency SLO, network and storage status, time estimation and I/O characteristics. The controller interacts with enforcers at client, network and storage nodes to coordinate their actions. Enforcers along the I/O path will adjust the priorities of I/Os accordingly. Software Defined Network(SDN) [130, 131, 132, 133] is integrated into JoiNS as part of the network enforcers to coordinate with storage. To avoid possible scalability issues, this logically centralized controller may have delegates on each node but function as a whole. When controlling I/O requests and responses, we distinguish reads from writes. We utilize the asymmetry property in read and write I/O packet size to better ensure the SLO of an I/O request with less penalty incurred on other traffic.

In this chapter, the following contributions are made:

- We identify the need to consider all the components along the I/O path from client to storage to ensure latency SLO.

- We design controller-based mechanisms to monitor the status of network and storage globally, estimate the latency of I/O requests and guide the control along the I/O path.
- We design an approach to control I/O packets with little overhead based on the asymmetry property in read and write.
- We build a real system to coordinate clients, network, and storage, and demonstrate the effectiveness of JoiNS in ensuring latency SLO.

The outline of this chapter is as follows: We first present the difference of JoiNS with related research in Section 4.2. Section 4.3 describes the motivation of our chapter and the challenges in solving the problem. In Section 4.4, we illustrate the design of JoiNS. The implementation of our proposed system is described in Section 4.5. We analyze the performance and verify the performance improvement of JoiNS in Section 4.6. Section 4.7 concludes the chapter.

4.2 Related Work

JoiNS is different from prior work in two main ways. First, it involves client, network and storage and considers the possible congestion at both network and storage. Second, it controls I/Os with best effort and focuses on the latency SLO compliance especially when there are components close to congestion.

Storage QoS or SLO has been studied for a while. Recently, as I/O stacks become more complicated, some research start to investigate QoS guarantee under different I/O stacks, including virtual machines [58], containers [113, 134], and networked storage [128]. In this chapter, we focus our discussion on the existing studies that try to involve both network and storage in networked storage environment. Recently, several researchers see the essence of including network when considering storage QoS, or vice versa. IOFlow [99] is an architecture that can enforce high-level flow policies. It allocates bandwidth for a flow from a particular VM to certain storage shares or routes an I/O traffic through a sanitization layer. It translates flow policies into queuing rules at individual stages along the path. In practice, it exercises control on SMB client and SMB server (SMB is a network file sharing protocol [135]), but it does not have

control on network that is between SMB client and SMB server. A congested network may break the SLO compliance they are trying to meet. JoiNS shares the same goal of meeting application's SLO requirements, but we tackle the scenario that both network and storage may become congested. In addition, JoiNS aims at solving the problem when resources are in shortage, especially at the boundary of getting close to congestion situation. IOFlow does not aim at this problem specifically. They share resources evenly among VMs whose requirements cannot be met.

sRoute [129] extends IOFlow's routing functions and is able to forward I/Os from over loaded servers onto less loaded servers. The forwarding is determined based on the queue size at each of the storage servers. The rerouting may change the traffic on network but it does not consider the network status.

PriorityMeister [128] tries to meet end-to-end tail latency SLOs by automatically and proactively configuring workload priorities and rate limits. It applies multiple rate limiters simultaneously for each workload at each stage. It uses a greedy algorithm to pick a priority ordering of these workloads statically. PriorityMeister assumes the system has full visibility and control over all workloads which may not be the truth in a networked storage environment. There is always traffic beyond our control including workloads from other users or data centers. In JoiNS, we monitor those traffic at real time. We react to the network and storage status change as workloads fluctuate. In JoiNS, priorities are dynamically assigned to each I/O rather than statically assigned in the granularity of workload. The valuable resources will be allocated to those I/Os really in need, especially when the system is close to congestion.

Pulsar [136] analyzes the demands of tenants and appliance capability, and allocate datacenter resources to meet their requirements. It estimates each tenant's resources based on its metric rather than trying to meet the SLO deadline. It focuses more on resource allocation on network and storage respectively, rather than have a thorough policy combining network and storage together.

Crystal [100] provides a framework to enforce policies of tenants including bandwidth SLO, and triggering storage functions (compression, encryption, caching) in object stores, e.g. OpenStack Swift [108]. Crystal is not designed for networked storage environment, thus it does not controls the network outside of OpenStack. It also needs exploration on ensuring latency SLOs of tenants.

4.3 Motivation and Challenges

In this section, we first discuss why it is important to coordinate all components along the I/O path. Then, we present the challenges in coordinating these components.

4.3.1 Why not individual control?

There are many mechanisms on alleviating the performance degradation of a single component. Controlling a single component and ignoring other components may not help the overall performance. For example, people may take the redundancy-based approach to tackle the long tail latency issue [137]. When the current network route where the I/O traffic is going is congested, the redundancy-based approach may replicate multiple requests and use multiple relay VMs to reach the same storage via different routes. The fastest response received will be returned to the client. However, if a large amount of I/O requests are duplicated, they will saturate the storage very quickly, thus making storage a bottleneck even though the delay on network is reduced. In case of storage congestion, the redundancy-based approach may send redundant requests to multiple replicas of the same object to seek for light loaded storage. This will bring about additional cost on network and may saturate some network routes. In fact, it is essential to be aware of the status of each component before taking any control actions.

Meanwhile, more performance improvement may be achieved by controlling multiple components together. Considering that in a networked storage environment, the network is congested and the latency SLO of a client is violated. There are several classical techniques trying to alleviate network congestion. For example, RED [138] will try to throttle the packet issuing rate of a client by dropping packets such that fewer packets can be delivered into network according to the TCP congestion control. ECN [139] tries to notify the client to reduce its transmission rate explicitly without dropping packets. We will demonstrate that we can achieve better latency performance by considering multiple components along the I/O path through an experiment. In this experiment, we generate some non-storage network traffic that interferes with the storage traffic of a client within a network. We refer to this additional traffic as "noise". The noise together with storage I/Os consumes 95% of the total network bandwidth along the I/O request path. The storage which has maximum 140MB/s throughput is light-loaded and is only

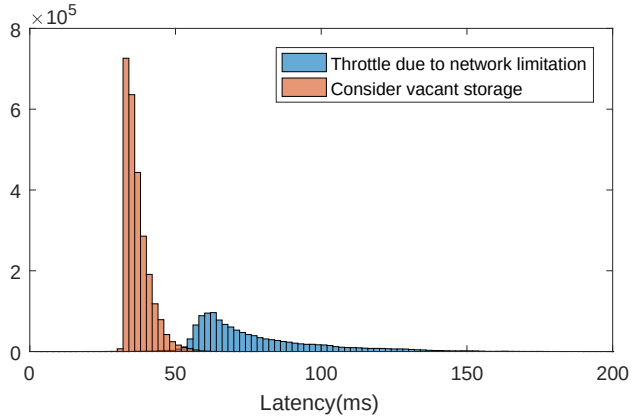


Figure 4.1: Histogram of IO latency. *Throttle due to network limitation* represents that I/Os are throttled due to congestion in network. *Consider vacant storage* represents that we prioritize I/Os to bypass the limitation in network because we know storage is light-loaded.

accessed by our client at a rate of 100MB/s. We set the SLO deadline to be 40ms. The latency we measured to meet the SLO is the time from the client issuing an I/O request until it receives the response from storage.

Under this workload and noise, there is a heavy load on the network request path. The client will throttle itself and issue fewer I/O requests to network to alleviate network congestion. Only 0.2% of the I/O requests meet the SLO with an average latency of 77.35 ms. However, if we let those I/O requests bypass the limitation imposed by the network and go to the light-loaded storage earlier by prioritizing those I/O requests along the network request path, 80.9% of the I/O requests can meet the SLO with an average latency of 36.95 ms. Figure 4.1 shows the histogram of I/O request latency in this experiment.

In summary, in a system involving multiple components like client, network and storage, a solution trying to guarantee latency SLO should not ignore any components. A careful consideration of all components and a global policy that coordinates those components along the I/O path can bring in more benefits for SLO oriented control.

4.3.2 Challenges

We highlight why it is difficult to coordinate components along the I/O path to guarantee SLO.

Global Visibility of I/O Stacks. In order to know when and where we should exercise control, we need to have a global view of the status of each component along the I/O path. However, it is challenging to acquire this global view since network and storage nodes are often remotely located and geographically distributed.

Coordination Between Components. There have been many solutions on controlling a single component to guarantee SLO. For examples, chunking I/O requests in storage [127, 140], differentiated scheduling in storage [127, 128, 136, 141], variants of weighted fair queueing in network [142], routing based on congestion in network [143, 144], throttling client [145] etc. However, coordination among multiple components requires understanding of the interactions among them. For examples, network needs to understand the I/O type and I/O size of an I/O request it is carrying; storage needs to understand network stacks. Since each component is only responsible for exercising control on that component, understanding the impact of controlling one component over other components becomes a must. Furthermore, such gap of interactions between network and storage also brings more difficulties to understand the status of all components as a whole and interpret them in a meaningful way to coordinate control. For example, the measurement of R/W throughput is usually used to show the storage status. But it does not look like anything to network switches, which usually use queue occupancy/length to indicate network congestion level.

SLO Aware. In order to ensure latency SLO, it is required that a control policy knows the exact latency requirement. Each request should know its SLO goal and can be adjusted to meet the goal at components along the I/O path. How to inform each I/O request and these components of the SLO is a problem.

Cost-effective Control. Any control imposed on the I/O requests will induce additional overhead on the performance. We need a control mechanism to ensure SLO with less overhead.

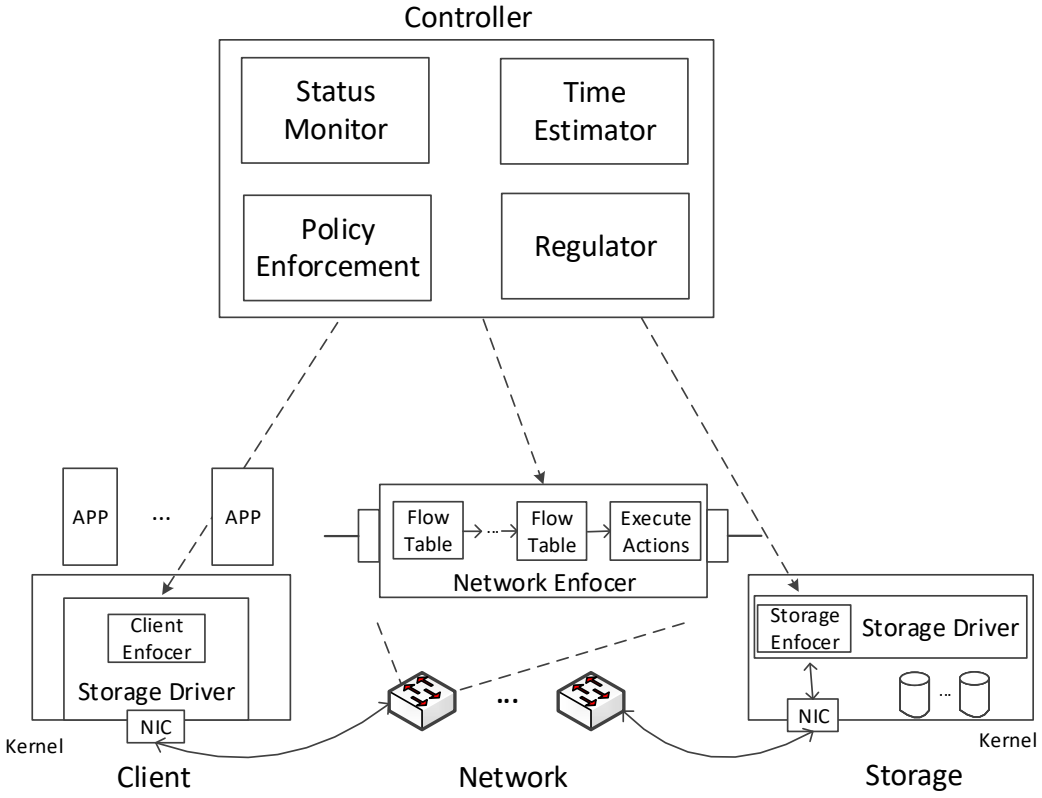


Figure 4.2: System Architecture of JoiNS.

4.4 Architecture

JoiNS ensures the latency SLO of I/O requests by coordinating different components along the I/O path.

4.4.1 System Design

Figure 4.2 shows the architecture of JoiNS. Client, network, and storage are monitored by a logically centralized controller. The controller keeps collecting the status data of each network and storage node via the **Status Monitor** module. These data are used to estimate the time needed for each I/O request at network request path, network return path, and storage in **Time Estimation** module. By comparing the estimated time with the required SLO, the controller can determine whether to control I/Os along the path in **Policy Enforcement** module. The actual latency of each I/O will feed

back into controller to refine the estimation in **Regulator** module.

Each component along the I/O path has an enforcer to control I/Os. The client enforcer controls admission of I/Os into network. Once an I/O request is delivered by NIC as I/O packets, they will enter their request path of network. The network enforcer at each network node can identify the packets that need control. The network enforcer implements differentiated scheduling through queues. Queuing rules are configured by the controller to dispatch I/O packets with different priorities into different queues. The priority of an I/O packet is set by each network enforcer according to the control decision made by the controller. In storage, once I/O packets arrive, they will first be reorganized as I/O requests. These I/O requests then will be processed by the storage enforcer. A differentiated scheduler is also implemented to differentiate I/O requests. Finally, they get processed by the storage device. The queuing rules in the storage enforcer are configured by the controller. How to determine the priority of each I/O in network and storage will be discussed in Section 4.4.6. Since it is I/O response generated by the storage that is delivered on the network return path, the control information related to an I/O request should be embedded into the response by the storage enforcer. On the return path, I/O packets will be matched again by network enforcers for possible control.

4.4.2 Status Detection

We design an algorithm for the status monitor module to have a global view of the network and storage status. We call it "probe and test". Considering any control imposed on I/Os will generate overheads, we will try to reduce the control. A control process will be wasteful when there is no congestion or little congestion in network and storage. They are also useless when the system is totally congested. Therefore, we need to determine the congestion level of the system first during runtime. This requires collecting information of the current network and storage status. Typically, the status information can be collected from on-going I/Os or by sending probes. Because the arrival of I/Os are unpredictable and may be intermittent, we choose to send probes to stably receive network and storage status. The status monitor module periodically sends active probes, which are specially designed I/O requests, to the storage and receives the responses. These probes collect data that can reflect network and storage status at real

time. The time interval between two rounds of probing is adjusted according to the variation of status change. In practice, the default time interval is 1 second. We use a sliding window to keep the history probing data. If the variation of data is too large, it means the system is not stable, thus reducing the time interval. If the variation is within a reasonable range, we increase the time interval gradually till the default 1 second.

The status monitor module will send out 3 probes in each round of probing: one read, one write and one storage query. To minimize the load imposed on network and storage, each read or write probe only contains the I/O request with the typically smallest I/O size, e.g., 4KB and they will not be actually processed by storage. Once received, the corresponding responses will be returned immediately by storage. The size of a read/write probe and its response will follow the actual size of the read/write request/response strictly. Each probe will be timestamped on entering the network, entering the storage, exiting storage, and arriving back at client. They are denoted as T_1, T_2, T_3, T_4 respectively. Hence the time spent on the request path and return path of the network for a read probe are calculated as $T^r_{rq} = T_2 - T_1$, $T^r_{rt} = T_4 - T_3$. We also have T^w_{rq}, T^w_{rt} for write probes. The storage will send back the current queuing time T_q in storage after receiving the storage query probe. All probes are assigned default priorities along the way.

By using these data, the controller can have an estimation of latency t_{est} for each I/O request. The controller will determine the congestion level for each I/O request by comparing its SLO (D) with the estimated latency. We set a congestion factor β ($0 < \beta < 1$) to define a safe zone for the latency. If $t_{est} \leqslant \beta D$, it means the current latency is safe and the system is **not congested** for this request. Hence, there is no need to do anything. If $\beta D < t_{est} \leqslant D$, it indicates that the system is **close to congestion** and there is a danger that the SLO will be violated. In this case, the controller determines to control this I/O request along the path. If $t_{est} > D$, it means the system is **fully congested**. In this case, we throttle the client. This congestion factor β can be adjusted by users of JoiNS. A small β indicates an aggressive control and the application is latency sensitive. A large β indicates relaxation on the SLO compliance. If β is set to 0, the proposed algorithm degenerates to *Pri_all* (defined in Sec. 4.6.2) which does not judge the congestion level for I/Os and prioritizes all I/Os of an application. Intuitively, it may be best to set β to be 0, but we will show that it

does not perform as well as JoiNS in Sec. 4.6.2.

4.4.3 Time Estimation

The controller estimates the time needed on network request path, network return path and storage for an I/O request based on the current network and storage status, as well as the I/O characteristics including I/O size and I/O type.

The network latency of an I/O can be characterized as the sum of transmission delay, propagation delay and queuing delay. We assume there are k hops from one end to the other, and the transmission speed on each link is $g_i (i \leq k)$. The MTU size of each link is $MTU_i (i \leq k)$. According to the packet switching theory [146], the latency for transferring x KB data from one end to the other is:

$$L = \frac{x}{g_{min}} + P + Q + \sum^{k} [\frac{MTU_i}{g_i} - \frac{MTU_i}{g_{min}}]$$

Where g_{min} is the minimum of all g_i. P is the propagation delay. Q is the queuing delay. Apparently, L is linear to x, and can be rewrited as $L(x) = a \cdot x + b$. a is the reciprocal of g_{min}. It can be directly calculated from g_{min} or calculated from multiple $(x, L(x))$ data points. b reflects the current network status (queuing delay plus other constants).

For a read probe, we have $T^r_{rq} = a_{rq} \cdot S_{\text{req}} + b_{rq}$, $T^r_{rt} = a_{rt} \cdot S_0 + b_{rt}$, where S_{req} is the packet size of a read request and S_0 is the I/O size of the probe. For an m KB read request, we can estimate the time on the network request path t^r_{rq} and the time on the network return path t^r_{rt}:

$$t^r_{rq} = a_{rq} \cdot S_{\text{req}} + b_{rq} \tag{4.1}$$

$$t^r_{rt} = a_{rt} \cdot m + b_{rt} \tag{4.2}$$

With each collected T^r_{rq} and T^r_{rt}, we can calculate the current network status b_{rq} and b_{rt}. But network is unstable and disruption is common. When a probe detects a network status change, we do not know whether it is a long time status change or just a temporary disruption. Thus, it is impossible to accommodate the estimation to every network status change. To make our estimation robust and as accurate as possible,

we use a moving window to smooth the network disruption. We calculate the moving average of b_{rq} and b_{rt}, and use them to calculate equation (4.1) and (4.2). The size of the moving window can be adjusted by users of JoiNS. A bigger window size is more robust to network disruption but takes more time to adjust to network changes. A smaller window is more sensitive to network changes but will cause big fluctuations on estimation caused by network disruptions. In practice, we set a window size of 10 in our evaluation and it achieves a good performance.

Similarly, we can calculate t^w_{rq} and t^w_{rt} for write requests from T^w_{rq} and T^w_{rt}.

Based on the service center model, the estimated time in storage can be calculated as $t^r_s = T_q + \frac{m}{Bandwidth_r}$ and $t^w_s = T_q + \frac{m}{Bandwidth_w}$. $Bandwidth_r$ and $Bandwidth_w$ are the storage bandwidth for read and write respectively. In general, we can rewrite them as

$$t^r_s = a_r \cdot m + b_s$$
$$t^w_s = a_w \cdot m + b_s$$

a_r and a_w are the reciprocal of $Bandwidth_r$ and $Bandwidth_w$ respectively. In a storage where read bandwidth is the same as write bandwidth, $a_r = a_w$. b_s reflects the current storage status(queuing delay). We also calculate the moving average of b_s to estimate the storage latency of an I/O.

We can calculate the estimated latency for an m KB read request t^r_{est} and for an m KB write request t^w_{est}:

$$t^r_{est} = t^r_{rq} + t^r_{rt} + t^r_s + \delta \tag{4.3}$$
$$t^w_{est} = t^w_{rq} + t^w_{rt} + t^w_s + \delta \tag{4.4}$$

Here δ is used by the Regulator module to amend the estimation against the actual latency.

In networked storage environment, the sizes of data transmitted on network request path and network return path for an I/O is very distinct. For example, if iSCSI is used, a 4KB read request only has 48B data to be delivered on network request path and has 4KB data to be delivered on network return path (more details will be discussed in Sec. 4.4.5). A typical I/O will access at least 512B of data. If a_{rq} and a_{rt} are

close in (4.1) and (4.2) (the g_{min} on network request path and return path are similar), transmitting a read response/write request will take 10x more time than transmitting a read request/write response on network. Thus, (4.3) and (4.4) can be simplified as:

$$t^r_{est} = b_{rq} + a_{rt} \cdot m + b_{rt} + t^r_s + \delta \tag{4.5}$$

$$t^w_{est} = a_{rq} \cdot m + b_{rq} + b_{rt} + t^w_s + \delta \tag{4.6}$$

We will discuss how we control I/Os with these estimated time in Section 4.4.6.

4.4.4 Estimation Refinement

When estimating latency for each I/O request, we use δ to amend the estimation. This δ is determined by the Regulator module according to the estimation and the actual latency. No matter whether equations (4.3) and (4.4) or the approximated equations (4.5) and (4.6) are used, the estimated latency of an I/O request with m KB IO size can be generalized as a function of a single variable: $t_{est} = a \cdot m + b + \delta$. a and b are constants. For each t_{est}, we have an actual latency t_{act}. We are going to calculate a δ so the squares of the residues of the actual latency and the estimated latency $D = \sum_i (t^i_{act} - t^i_{est})^2$ is minimized. By applying least squared method, we are able to calculate

$$\delta = \overline{t_{act}} - \overline{a \cdot m + b}$$

The average of t_{act} and $a \cdot m + b$ is calculated on the last n requests. If n is large, δ will make the estimation more robust but less adaptive to network and storage status change. If n is small, δ will make the estimation more sensitive to network and storage status change but cause big fluctuations on estimation caused by network or storage disruptions. In practice, we set n to be 20 in our evaluation and it achieves a good performance.

With amending the estimation, not only can we adjust to network and storage status change quickly, but also prioritize those I/Os in need and cancel unnecessary prioritization. If the estimated latency of an I/O meets $t_{est} \leqslant \beta D$ and that I/O is not prioritized, it is possible that the actual latency falls into $(\beta D, D]$ or even greater than D. If this case happens continuously, it will cause δ to increase and further increase the

estimated latency. In turn, more I/Os will be prioritized. If the estimated latency of an I/O meets $\beta D < t_{est} \leqslant D$ and that I/O is prioritized, it is possible that the actual latency is not greater than βD. This case indicates the effect of prioritization may be too aggressive for that I/O. If this happens continuously, it will cause δ to decrease and fewer I/Os will be prioritized. In summary, by adjusting δ, JoiNS is trying to only prioritize those I/Os that are in danger of SLO violation and reduce unnecessary prioritization at real time.

4.4.5 Distinguish Read from Write

Our special control of I/Os is based on the asymmetry property existing in read and write I/O packet size. In a networked storage environment, the I/O requests and their responses are encapsulated in I/O packets on network. Traditionally, network functions like routing and switching do not treat these I/O packets differently and hence ignore the intrinsic characteristics inside storage I/Os.

On the network request path, a read request contains only I/O request while a write request contains the data to be written to the storage. On the other hand, on network response path, a read response contains the data that the client requests while a write response is just a notification of success or failure of the write. Taking an example of iSCSI, a read request for reading 4KB of data has only 48B data including a 16B SCSI command encapsulated in a single network packet. On the other hand, a write request writing 4KB data includes not only a write SCSI command but also the 4KB data to be written. Since the size of data to be written is typically much larger than one MTU size, a write request will be broken into multiple network packets. On the response path, this asymmetry reverses. The read response contains 4KB data spanning across multiple network packets. The response of that write request is a 48B iSCSI response indicating write success or failure.

Since we want to ensure the SLO of each I/O, we may choose to prioritize all network packets composing that I/O request and its response. However, prioritizing these I/O packets is sure to delay other packets. Considering the asymmetry in I/O read and write, we distinguish reads from writes. On the request path, we choose to prioritize read requests. On the return path, we prioritize write responses due to their smaller size.

4.4.6 Coordination

JoiNS coordinates client, network and storage to ensure the latency SLO of I/Os.

Interactions between storage and network. In order to coordinate network and storage, how network can understand the interactions with storage and vice versa becomes the first problem. In a networked storage environment, an I/O request is delivered in one or multiple network packets on network. They are then extracted out from network packets at storage and the network packet headers are discarded. Considering network preserves all storage information while storage loses network information, all control information sent to storage should be incorporated in the command (e.g., SCSI command) of the I/O request and all control information sent to network can be incorporated in network headers.

Admission control. Once the controller determines the congestion level for an I/O through probe and test algorithm, it will determine whether to control this I/O. If the system is not congested, this I/O does not need any control. The client enforcer will admit that request directly into network. If the system is tested to be close to congestion, the controller will control this I/O. The client enforcer will mark that I/O in the corresponding network packet headers and the storage command. If the system is tested to be fully congested, the client enforcer will also deliver that request into network and let the client be throttled by TCP congestion control.

Prioritize I/Os. Once I/O requests are admitted into network, the priorities of I/Os will be adjusted at each network node and the storage node. If an I/O is marked by the controller, network enforcers can recognize it by matching the corresponding I/O packets against the queuing rules. According to the matched rules, a priority will be set for each I/O packet. Then, each packet will be dispatched to the queue matching the priority. The storage enforcer will also set the priority of the I/O request extracted from the received I/O packets, and dispatch it to the queue matching the priority. In our priority algorithm, we set up two queues (fast and regular) in each network and storage node. The fast queue indicates high priority, and the slow queue indicates low priority. Once an I/O packet is marked in the network header, it will be assigned to the fast queue in network. Once an I/O request is marked in the storage command, it will assigned to the fast queue in storage. People can develop more priorities and more queues with different speed in network and storage nodes to enrich the priority

algorithm.

4.4.7 Scalability

The controller of JoiNS is logically centralized. The functions of controller are actually distributed to every client. The controller daemon on each client will probe network and storage, estimate latency, determine to control I/Os and refine estimation when accessing its remote storage. The daemon on each client is a delegate of the controller. Since each client may have multiple storage and go through multiple network routes, maintaining a status map by sending probes to all routes and storage is cumbersome. Besides, monitoring network and storage status is wasteful when the client is vacant. We will only launch those controller functions at client when the client starts to connect with storage, signaled by the first I/O request or TCP handshake packets. Then the controller daemon will only probe the storage that this client is accessing and probe the network those I/Os are going through. After the client is idle for a period (e.g., 60 seconds) or TCP disconnects, the controller daemon will stop the functions. By these means, JoiNS is able to scale easily and reduce the burden on network and storage as much as possible.

4.5 Implementation

Our implementation is based on iSCSI. Our storage is connected through Ethernet and IP networks to clients. A client accesses its storage via an iSCSI initiator to the iSCSI target (networked storage device) at a remote location.

Client enforcer is responsible for admitting I/O requests into network. We implement and insert a hook inside iSCSI initiator. If the controller determines that a read request needs further help, the client enforcer will tag the packet of that read request with DSCP flags which can be recogonized by network enforcers. It also sets a miscellaneous byte of iSCSI CDB [147] such that this request can be recognized by storage enforcer. If a write request needs further assistance, it only sets the miscellaneous byte of iSCSI CDB. After the iSCSI target receives it, the storage enforcer will tag the response of that write request with DSCP flags so that it can be recognized on the return path of the network. Once the controller finds that the estimated latency of a request is greater

than the SLO, the client enforcer will just deliver that request into network and the client will be throttled according to the TCP congestion control.

Network enforcer is responsible for initiating and dynamically configuring priority queues in programmable switch. Pioneers like [148, 149, 150, 151] have tried to optimize I/Os on network devices with network functions, e.g., data deduplication [152, 34], etc. Our network enforcers also control I/Os in network. In our prototype implementation, we utilize Openflow virtual switch [133, 153] (OVS) as the programmable switch (an SDN implementation). Please note that JoiNS does not rely on Openflow-capable switch. Legacy switches which provide programmability interface can also be integrated into JoiNS.

OVS works in a paradigm that control plane is decoupled from the underlying data plane. It provides high programmability that enables configuring and controlling network dynamically. For example, a software program can act as a network controller to interact with OVS by configuring forwarding rules and reacting to topology and traffic changes. The forwarding rules in OVS is typically flow-based, and thus it inherently supports matching network packets against flow entries and taking corresponding actions. The controller uses OpenFlow protocol to communicate with OVS. Flow entries can be installed proactively or reactively by the controller. When the first packet of a new flow arrives, the switch will look for a matching entry in the flow table. If there is no matching entry, the OVS notifies the controller which will install corresponding rules into OVS. The OVS version we use in the prototype is 2.0.2, and the OpenFlow protocol follows version 1.3.

In our implementation, we attach two queues on each output port of an OVS. Both queues attached to a port have the same maximum rate equal to the link rate of that output port, but have different minimum rates. A queue with a higher minimum rate has a higher priority. We configure one queue with 0 minimum rate as regular queue, and the other queue with minimum rate equal to link rate as fast queue. When the network is light-loaded, we only use the queue with minimum rate of 0. Because there is no contention from the other queue, it functions as if there is no priority differentiation at all and can take the full link bandwidth of the output port. We implement a control application that will determine the priorities of I/O packets and install flow entries into Open vSwitches. Each OVS will first match I/O packets against the flow entries to

check whether the DCSP flag is marked. If an I/O packet is marked, it means the packet needs control. Then the packet will be forwarded to other flow tables inside the switch to match flow entries further to determine its priority inside this network node. Finally it will be forwarded to the fast or regular queue attached to the output port.

Storage enforcer is responsible for differentiating and scheduling I/O requests. Our current implementation inserts a hook inside iSCSI target. Once an I/O request is admitted, the iSCSI target checks the miscellaneous byte in iSCSI CDB. If it is set, it determines the priority of the request and dispatches it to the fast or regular queue. In addition, the iSCSI target will take responsibilities of marking write responses by tagging the packets with DSCP flags if the iSCSI CDB miscellaneous byte is set in the corresponding write requests.

4.6 Evaluation

This section evaluates JoiNS. Part A presents the setup of experiments. We demonstrate the ability of JoiNS in meeting latency SLOs in a networked storage environment in Section 4.6.2. Section 4.6.3 studies the characteristics of JoiNS as network and storage load changes. In Section 4.6.4, we show how JoiNS performs to multiple clients compared with other polices. Section 4.6.5 evaluates the cost of prioritizing read requests and write responses utilized in JoiNS. Section 4.6.6 shows that JoiNS still works well when the time estimation is inaccurate. Overall, our key findings are:

- Using trace-driven experiments, we show that JoiNS can ensure applications' latency SLOs with little overhead. JoiNS helps most when the system is close to congestion and does little help when the system is not congested or fully congested.
- JoiNS strikes a balance between latency SLO ensuring and the overhead caused by the control on I/Os.
- When there are multiple clients competing for network and storage resources, JoiNS works well in meeting latency SLOs for all clients.
- JoiNS is robust to inaccurate latency estimation and robust to bursty workloads.

Table 4.1: Workload traces used in our evaluation.

Workload label	Workload Description	Interarrival Variance
Workload A	MSR web staging stg_1	86.68
Workload B	MSR research projects rsrch_0	12.17
Workload C	MSR user home directories usr_0	6.55
Workload D	MSR print server prn_0	12.76
Workload E	MSR firewall/web proxy prxy_0	22.85
Workload F	Synthetic 100MB/s read I/Os	-
Workload G	Synthetic 70MB/s read I/Os, 30MB/s write I/Os	-
Workload H	Synthetic 50MB/s read I/Os, 50MB/s write I/Os	-
Workload I	Synthetic 20MB/s read I/Os	-

4.6.1 Experiment setup

Our testbed comprises 5 servers, each with 24 Intel Xeon 2.40GHz E5-2620 v3 cores and 64GB of memory. Each server has four 1Gb/s Broadcom NetXtreme BCM5720 NIC ports, connected to a HP ProCurve 5406zl switch. They are all running Ubuntu 14.04. One server acts as a client that sends I/O requests to storage. Two servers act as storage nodes. One of them serves as iSCSI target and provides storage and the other serves as the storage proxy. They use one HDD volume with maximum bandwidth of 120MB/s as the backend. The remaining two nodes serve as SDN switches with the link speed of 1Gb/s.

Datasets. We evaluate our system using a collection of real block I/O [154] and synthetic storage traces. Table 4.1 summarizes all the workloads we use in the evaluation. For real block I/O traces (Workload A - E), we also show the squared relative standard deviation of the interarrival time, which describes the burstiness of the workload. A higher squared relative standard deviation indicates more bursty arrival patterns. With different interarrival variances, we show our system can adapt to different burstiness.

Policies. In our experiments, we use 5 approaches to understand different aspects of the performance of JoiNS. *JoiNS* is our primary mechanism that tries to ensure latency SLO with little overhead. It is able to adapt to network and storage status change. In our experiment, we set the congestion factor β to be 0.8 when the controller determines

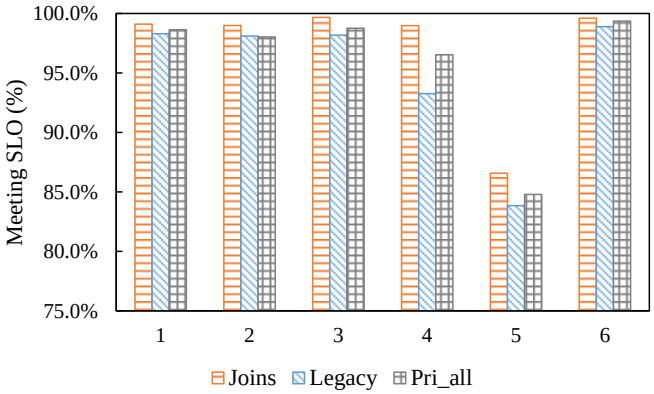


Figure 4.3: Percentage of I/O requests meeting latency SLO

the congestion level for each I/O. In the *Legacy* system, all components are in default configurations. That is, FIFO in network and storage. In *Pri_all* approach, we do not judge the congestion level for I/Os but prioritize all read requests and write responses of an application. In *Pri_both* approach, if an I/O needs to be prioritized, both the request and the response will be prioritized. We use this approach to evaluate the performance of distinguishing read from write. We implement the policy in the PriorityMeister [128] chapter as *PM* to evaluate the latency SLOs for multiple latency sensitive workloads. *PM* works in the system which has full knowledge and control of workloads. It analyzes the representative trace proactively for each workload and applies the r-b pairs (rate and token bucket size) to the workloads which enable workloads run in an unfettered manner. It uses a greedy algorithm to pick a priority ordering of the workloads. In the experiment, we first analyze the trace we are replaying and generate the r-b pairs used in token bucket for each workload. With predefined latency SLO, we use the prioritizer algorithm described in PriorityMeister to configure the priorities of workloads.

4.6.2 JoiNS latency performance

In this subsection, we demonstrate the ability of JoiNS in meeting latency SLOs in a networked storage environment. Figure 4.3 plots the latency performance across Workload A - E. In this experiment, we replay each trace based on timestamps for one hour. Meanwhile, we generate competing network traffic and storage traffic (as noise) on both request path and return path. By analyzing the throughput of each workload, we are

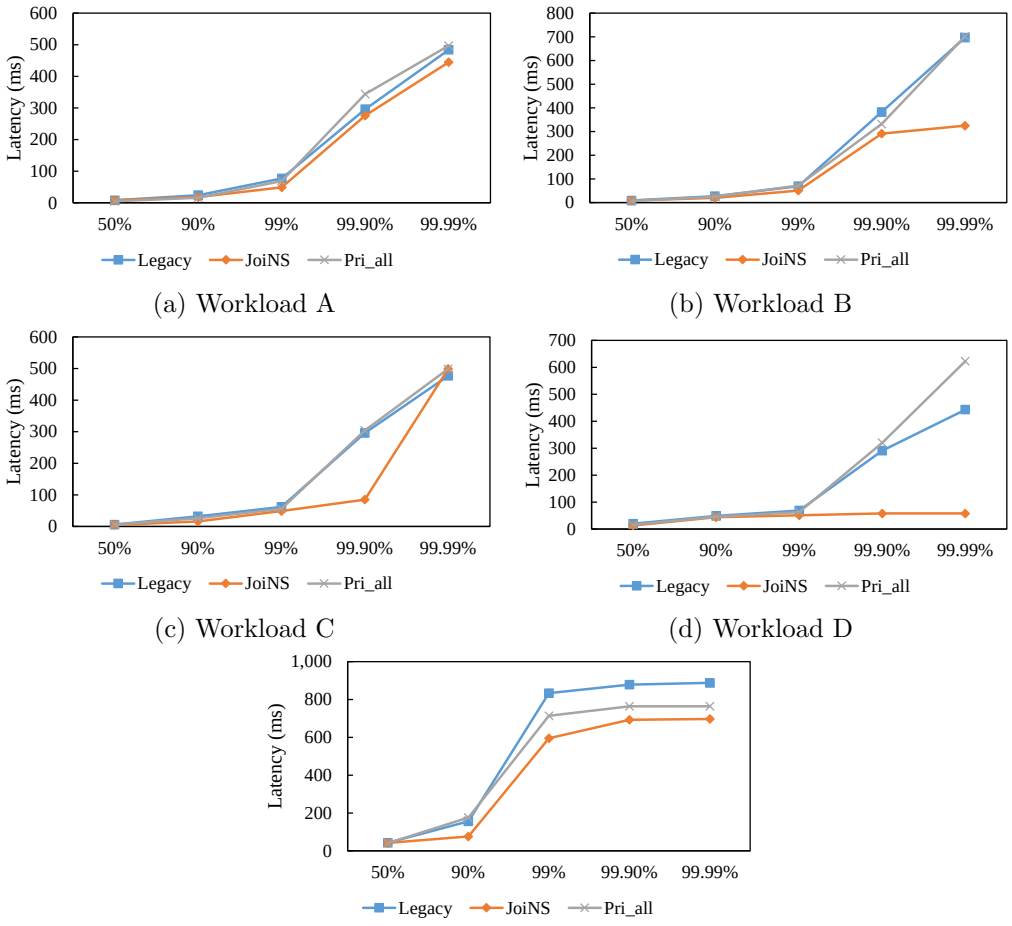


Figure 4.4: Request latency at different percentiles.

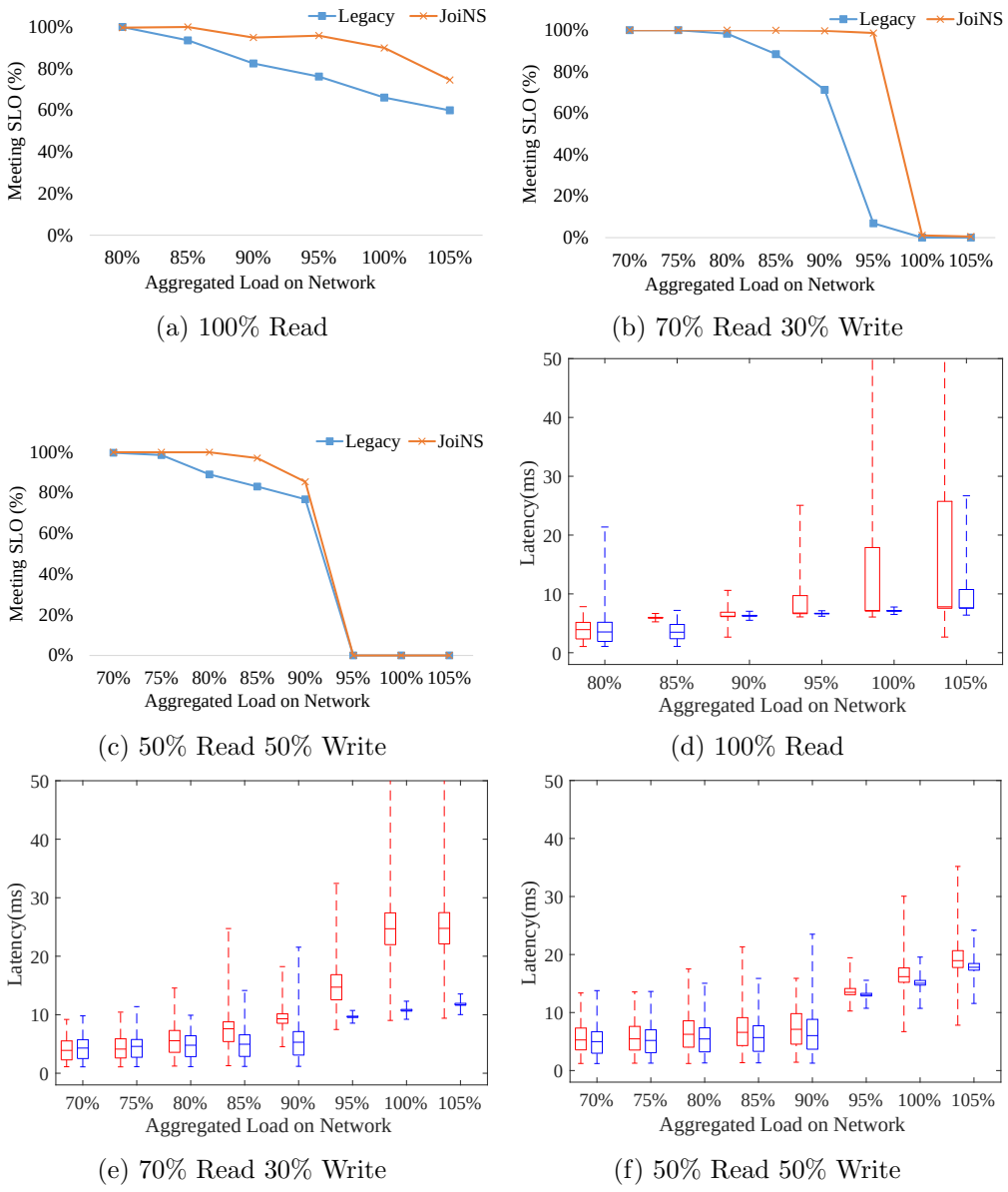
able to generate proper noise so that the aggregated load transmitted on network and storage are around 95% of the total bandwidth, which is close to congestion. We set latency SLO as 50 ms for these workloads. We can see there is a higher percentage of I/Os meeting SLO for each workload with *JoiNS* than *Legacy* and *Pri_all*. It is interesting to see that simply prioritizing all traffic of a workload without the mechanisms of JoiNS does not performs as well as JoiNS. This is because those I/Os which deem the system as not congested or fully congested are also prioritized. Controlling these I/Os unnecessarily will bring additional computation overhead in network and storage nodes and also occupy more resources which should have been allocated to other I/Os.

Figure 4.4 shows a more descriptive representation of the latency at different percentiles. The results are grouped by workloads. For each workload, we show the tail latency at 90th, 99th, 99.9th and 99.99th percentiles. At 90% and 99% tails, JoiNS can meet SLOs and has smaller latency than the other 2 approaches across traces A-D. For longer tails at 99.9% and 99.99%, although JoiNS still achieves smaller latency, the SLOs are violated. For workload E, all policies violate SLOs after 90% tail but JoiNS has smaller latency at all tails. The reasons are twofold. First, our control is based on the estimation of latency. Any ways of estimation have bias and result in controlling or not controlling some I/Os by mistake. Second, when the system is fully congested for an I/O request, we will not control that I/O. Therefore, those I/Os which will violate SLOs for sure will still violate SLOs.

If we define a workload is bursty when its squared relative standard deviation of the interarrival time is greater than 20, we have workload A,E as bursty workloads and B,C,D as not bursty workloads. Besides, the interarrival variance of workload A is 3-4 times of workload E. We can define workload A as extremely bursty. We can see from Figure 4.3 and 4.4, our JoinS performs well over Legacy and Pri_all for workloads with different burstiness. For workload A, the improvement on the percentage of I/Os meeting SLOs and the decrease of tail latency are not that obvious compared with other workloads. This is because a higher burstiness increases more queueing and the latency of a workload as well.

4.6.3 Reactions on network and storage status change

In this subsection, we study the characteristics of JoiNS to better understand the strength and limitation of JoiNS. In the experiment, we change the network and storage status. Figure 4.5 shows the performance of JoiNS when network starts to become congested and storage is light-loaded. Because the throughput of workload A-E is not stable, using synthetic workload which has stable throughput will make it easier to control the load on network and storage. We run workload F,G,H in this experiment. In each workload, the aggregation throughput of I/Os is 100MB/s, with I/O request size of 64KB. We try different combinations of read and write ratio and controlled read I/Os while leaving write I/Os for observation. At the same time, we have competing network traffic and we consider them as noise. We generate network noise on both



(a) 100% Read

(b) 70% Read 30% Write

(c) 50% Read 50% Write

(d) 100% Read

(e) 70% Read 30% Write

(f) 50% Read 50% Write

Figure 4.5: I/O latency when network starts to become congested while storage is light-loaded. Graph (a)-(c) shows the percentage of requests meeting latency SLO. Graph (d)-(f) plots the latency distribution of I/Os. At each point of aggregated load, the left box represents legacy system and the right box represents JoiNS.

request path and return path, and ensure the aggregated load transmitted on these paths are the same with each other. The read write ratio in these 3 workloads are 100%

Read, 70% Read+30% Write and 50% Read+50% Write respectively. The aggregated load including I/Os and noise on network varies from 70% to 105% of the total network bandwidth (x-axis). In 100% Read workload case, we already have 22.27 Mb/s of data (read request) on request path and 800Mb/s of data (read response) on return path without any noise. Therefore, the minimum aggregated network load for this workload is 80%. That is why the x-axis starts from 80% in Figure 4.5(a) but starts from 70% in Figures 4.5(b) and (c). As to the workloads with more writes than reads, its load on network is symmetric to the case with more reads than writes. Thus the performance will be exactly the same as what we are showing here.

From Figures 4.5(a)-4.5(c), we can see JoiNS outperforms the Legacy in terms of percentage of requests meeting SLO. Taking the 70% read case for example, when the network is light-loaded (70%-80% aggregated network load), JoiNS performs the same as the Legacy system. This is because the performance is already good enough when the network is not congested. As the noise increases (at 80%-95%), the legacy system performance declines. At early stage (80%-90%), the congestion level is not that harmful so the performance declines slowly. However, at 90%-95% level of load, the system becomes close to congestion for most I/Os and the performance declines sharply. By comparison, the performance is still stable in JoiNS at the same congestion level. As we increase the network noise further, our JoiNS drops dramatically after 95% and then keeps low, while the performance of Legacy keeps low all the time. This is because the network becomes fully congested. JoiNS can help a little bit in a fully congested system.

Such a pattern of performance change can also be verified from latency distribution of I/Os from Figures 4.5(d)-4.5(f). We can see there is an obvious transition point (95% aggregated network load in 70% Read+ 30% Write case) where the median latency of legacy system increases slowly before this point and climbs high after this point. At the same time, the median latency of JoiNS keeps stable before this point and has a jump as well at this point (but lower than Legacy). This point is where the system is close to congestion from the application point of view.

Meanwhile, we study the performance of JoiNS when storage starts to become congested while network is light-loaded. Since storage has roughly the same performance for reads and writes, there is no need to experiment with different read/write ratios. In

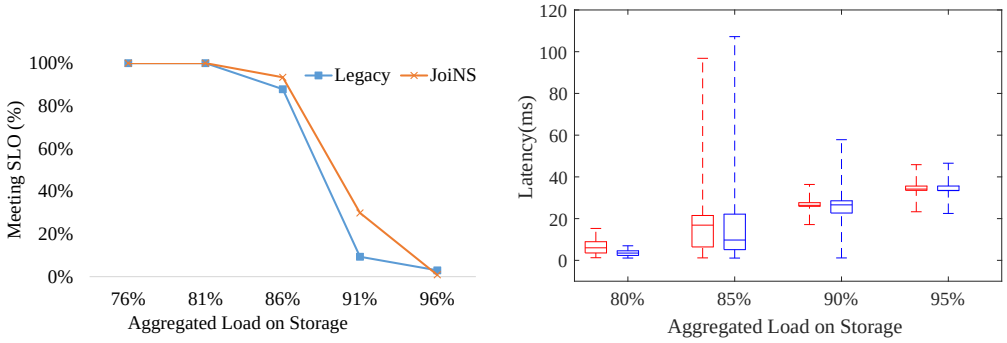


Figure 4.6: I/O latency when storage starts to become congested while network is under-loaded.

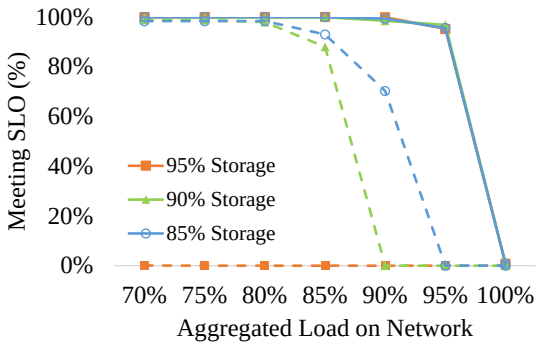


Figure 4.7: I/O latency when network and storage are in different status. (solid - JoiNS, dashed - Legacy)

this experiment, we run workload I with 64KB I/O size. At the same time, we have competing storage I/Os as noise. Figure 4.6 shows similar properties. The x-axis is the total consumed storage bandwidth by both generated I/Os and noise. We can see when the storage noise level is low (total load less than 80%), JoiNS has similar performance as the legacy system. As the noise increases, the I/O performance in the legacy system starts to decline and then drops dramatically at close to congestion status (85%-90%). It finally keeps at a low level after 95% (fully congested). The I/O performance in JoiNS drops at close to congestion status too, but still outperforms the legacy system. When the system is fully congested, even JoiNS cannot help. The boxplot showing the I/O latency further verifies such behaviors.

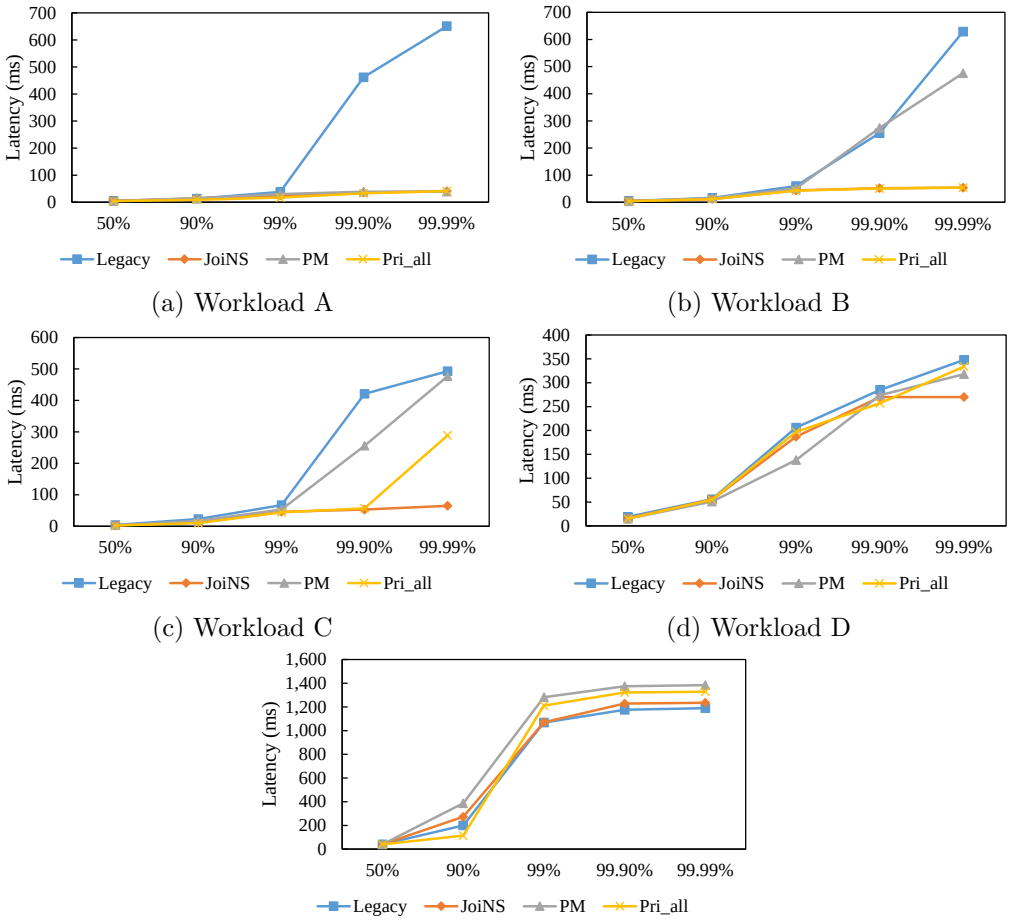
Figure 4.7 shows the performance of JoiNS when we have both network and storage noise trying to saturate the network and storage at the same time. Each line shows the percentage of requests meeting latency SLO from workload G. Different colors of lines represent different aggregated storage loads. Based on Figure 4.6, here we only need to experiment with 85%-95% aggregated storage load which is the range that I/O requests have obvious performance decline. The dashed lines show the latency in the legacy system and the solid lines show the I/O performance in JoiNS. Under the same storage load (any two lines with the same color), the solid lines show an obvious better performance than dashed lines. Especially when the network is at close to congestion status (80%-95%), the performance of JoiNS only drops little (<5%) and still maintains a high SLO compliance, while the legacy system drops more than 75% and almost no requests meet SLO. Under the same network load, when the storage is at close to congestion status (85%-90%), JoiNS also shows significant performance improvement than the legacy system.

From these experiments, we can conclude that JoiNS helps most when the system is close to congestion and does little help when the system is not congested or fully congested for both network and storage.

4.6.4 Multiple clients

In a typical networked storage environment, there are multiple clients sharing network and storage resources. We perform an experiment running workloads A-E together. Each workload represents a client. All the workloads share network and storage resources. Meanwhile, we generate competing network traffic and storage traffic (as noise) to make the network and storage close to congestion. According to the load on network and storage in nature, we set a trace a higher latency SLO if it has a higher load. In this experiment, we set latency SLOs of workload A-E to be 35ms, 40ms, 45ms, 50ms and 55ms respectively. In JoiNS, the priority of each I/O is set at real time based on the system status and latency SLO. In PM, the priority ordering is selected as A,B,C,D,E from high to low.

Figure 4.8 shows the tail latency for these workloads by applying different policies. It is essentially a representation of CDF at different percentage of latency. The results are grouped by the workload and it is easy to see JoiNS outperforms other policies for



(a) Workload A

(b) Workload B

(c) Workload C

(d) Workload D

(e) Workload E

Figure 4.8: Request latency of workloads running at the same time at different percentiles.

all workloads. Comparing with PM, JoiNS only prioritizes those I/Os whose estimated latency falls between βD and D. But in PM, once a priority of a workload is determined, all I/Os will go through the queue with that particular priority in network and storage regardless of network and storage status. Further, JoiNS only prioritizes read requests and write responses to reduce overhead to other I/Os while PM also controls read responses and write requests. For workload E, none of the policies improve much on tail latency compared with other workloads, as it has high latency in nature. JoiNS does not control I/Os that will violate SLOs anyway.

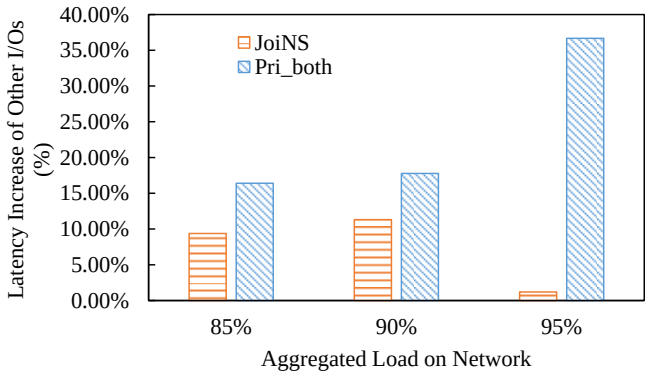


Figure 4.9: The cost of prioritizing based on the asymmetry in read and write I/O packet size compared with prioritizing both the request and the response of an I/O.

In general, JoiNS selectively prioritizes I/Os based on the SLO and estimated latency. The fine grained control of JoiNS imposes little overhead to each other when multiple clients are running together. In comparison, PM is easier to setup but requires providing representative traces in advance or waiting for the workload running for a while. It implies that if it is not stringent on ensuring latency SLO, an easy setup provided by PM will be a good choice.

4.6.5 Prioritization Cost

Our control is based on the asymmetry property in read and write I/O packet size, and we only prioritize read requests and write responses to generate a small overhead. In this experiment, we evaluate the cost of this approach. When we prioritize some packets, other packets will be delayed and their latency is sure to increase. The increase of latency of other packets is defined as prioritization cost.

Here we use the same experiment setting as the experiment in Figure 4.5. We use workload G and control read I/Os while leaving write I/Os for observation. We compare the percentage of latency increase of write I/Os in JoiNS with Pri_both.

Figure 4.9 shows the cost when the network is close to congestion. It is obvious that the cost of JoiNS is much less than Pri_both. We know from Figure 4.5(e) that the latency of the controlled I/Os jumps at 95% aggregated load. We see the same pattern in this experiment. When the aggregated load is 95%, the latency of I/Os jumps much

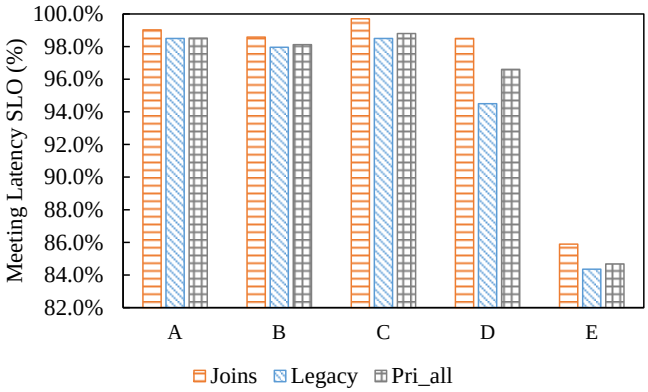


Figure 4.10: Percentage of I/O requests meeting latency SLO on latency mis-estimation.

larger than the cases with lighter load. That is why the percentage of latency increase in JoiNS becomes smaller at 95% aggregated load in Figure 4.9 compared with that at lighter loads.

4.6.6 Sensitivity Analysis

Latency estimation on network and storage are known to be challenging problems, and we are interested in JoiNS's performance if the estimation is inaccurate. In this experiment, we replace our time estimation with an inaccurate estimation that sets b_{rq}, b_{rt} and b_s to be constant 0 in t^r_{rq}, t^r_{rt},t^w_{rq}, t^w_{rt}, t^r_s and t^w_s. Figure 4.10 shows the same experiment as in Sec. 4.6.2, but with less accurate latency estimation. It is obvious that JoiNS still produces similar latency results as in Figure 4.3.

Although having more accurate latency estimation for I/Os is better, JoiNS does not rely on having accurate estimation. This is because the Regulator module is able to compensate for mis-estimation based on the actual latency observed. But it does not mean estimating latency is not important. As Regulator's adjustment is based on the real latency and having delay, if the workload or network and storage status is frequently changing, this delay will cause a large number of mis-controls.

4.7 Conclusions

This chapter present JoiNS, a system that coordinates different components along the I/O path to ensure latency SLO in an environment where clients access remote storage through network. JoiNS includes client, network and storage into control. It deploys a logically centralized controller to coordinate the control on each component. The controller has a global visibility of network and storage. It monitors the status of network and storage, and estimates the time needed on network and storage respectively for each I/O request. It determines whether to control I/Os along the path based on the SLO deadline, network and storage status, time estimation and I/O characteristics. Enforcers along the I/O path will adjust the priorities of I/Os accordingly. JoiNS is adaptive to the system status change and will only exercise control when there is a need. Through experiments, we show JoiNS is able to improve SLO compliance significantly.

In the future, we consider to apply the methods in sRoute [129] and reroutes I/Os when there is congestion. We will explore how to integrate this method into JoiNS to make it adaptive to the system status change. We will also explore more methods of control in OpenStack Cinder [155] and Swift [108] and other cloud storage environment.

Chapter 5

Conclusion

This thesis focuses on improving data access performance of applications in the hyper-converged infrastructure. We mainly focus on three different issues in the emerging hyper-converged infrastructure to improve the data access performance.

In Chapter 2, we identify the storage requirements of one prevalent virtual machine application, Virtual Desktop Infrastructure (VDI). We propose a system model to describe the I/O behavior of both homogeneous and heterogeneous configurations of VDI. By collecting VDI traces and plugging the traces to the model, we identify the storage requirements of VDI and determine the bottlenecks on specific target virtual disks at a specific time. Based on this information, we can tell what capacity and minimum capability a storage system needs in order to support and satisfy a given VDI configuration. We show that our model can describe more accurate and fine-grained storage requirements of a VDI system compared with the rules of thumb currently used in industry.

In Chapter 3, we design and implement k8sES (k8s Enhanced Storage), that efficiently supports applications with various storage SLOs along with all other requirements deployed in the Kubernetes environment based on Docker containers. In k8sES, we allow users to put their detailed storage requirements directly in their configuration files. We enhance the current Kubernetes scheduling process to simultaneously select storage and hosts. After selecting an appropriate host and storage, k8sES will carve out storage resources automatically from the selected storage on the fly. The storage

allocation mechanism in k8sES also improves the storage utilization efficiency. During the runtime of applications, k8sES can monitor the performance of both pods and storage, and adjust the storage resource allocation based on the storage SLOs compliance. The evaluation shows that k8sES can better meet users' storage SLOs along with other requirements with little effort from users and administrators. At the same time, k8sES can achieve a higher resource utilization efficiency with overhead similar to that of the current Kubernetes.

In Chapter 4, we propose and implement JoiNS, a system trying to guarantee latency SLOs for applications that access data on remote networked storage. We identify the need to consider all the components along the I/O path from client to storage to ensure latency SLOs in networked storage environment. JoiNS has both global network and storage visibility with a logically centralized controller which keeps monitoring the status of each involved component. JoiNS coordinates these components and adjusts the priority of I/Os in each component based on the latency SLO, network and storage status, time estimation, and characteristics of each I/O request. We design an approach to control I/O packets with little overhead based on the asymmetry property in read and write. We integrate Software Defined Network (SDN) into our system to coordinate with storage. Our evaluation shows that JoiNS can ensure applications' latency SLOs with little overhead in the networked storage environment.

References

[1] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, pages 164–177, 2003.

[2] Anthony Velte and Toby Velte. *Microsoft virtualization with Hyper-V*. McGraw-Hill, Inc., 2009.

[3] Lxc. `https://help.ubuntu.com/lts/serverguide/lxc.html`. Accessed: 2019-2-26.

[4] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[5] Amazon elastic block store. `https://aws.amazon.com/s3/`.

[6] Amazon s3. `https://aws.amazon.com/s3/`.

[7] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[8] Li Zhou, Ren Chen, Yinglong Xia, and Radu Teodorescu. C-graph: A highly efficient concurrent graph reachability query framework. In *Proceedings of the 47th International Conference on Parallel Processing*, page 79. ACM, 2018.

[9] Li Zhou, Yinglong Xia, Hui Zang, Jian Xu, and Mingzhen Xia. An edge-set based large scale graph processing system. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 1664–1669. IEEE, 2016.

[10] Google supercharges machine learning tasks with tpu custom chip. `https://cloud.google.com/blog/products/gcp/google-supercharges-machine-learning-tasks-with-custom-chip`. Accessed: 2019-2-26.

[11] Li Zhou, Hao Wen, Radu Teodorescu, and David H.C. Du. Distributing deep neural networks with containerized partitions at the edge. In *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19)*, Renton, WA, 2019. USENIX Association.

[12] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *ACM SIGARCH Computer Architecture News*, volume 45, pages 615–629. ACM, 2017.

[13] Hyuk-Jin Jeong, Hyeon-Jae Lee, Chang Hyun Shin, and Soo-Mook Moon. Ionn: Incremental offloading of neural network computations from mobile devices to edge servers. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 401–411. ACM, 2018.

[14] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[15] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, et al. Ese: Efficient speech recognition engine with sparse lstm on fpga. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 75–84. ACM, 2017.

[16] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 243–254. IEEE, 2016.

[17] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 6848–6856, 2018.

[18] Meng Yang, Bingzhe Li, David J Lilja, Bo Yuan, and Weikang Qian. Towards theoretical cost limit of stochastic number generators for stochastic computing. In *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 154–159. IEEE, 2018.

[19] M Hassan Najafi, S. Rasoul Faraji, Bingzhe Li, David J Lilja, and Kia Bazargan. Using resolution splitting to enhance performance of deterministic bit-stream computing. In *2018 27th International Workshop on Logic and Synthesis (IWLS)*, 2018.

[20] Bingzhe Li, Yaobin Qin, Bo Yuan, and David J Lilja. Neural network classifiers using a hardware-based approximate activation function with a hybrid stochastic multiplier. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 15(1):12, 2019.

[21] S Rasoul Faraji, M Hassan Najafi, Bingzhe Li, Kia Bazargan, and David J Lilja. Energy-efficient convolutional neural networks with deterministic bit-stream processing. In *Design, Automation, and Test in Europe (DATE)*, 2019.

[22] M. H. Najafi, S. Rasoul Faraji, B. Li, D. J. Lilja, and K. Bazargan. Accelerating deterministic bit-stream computing with resolution splitting. In *2019 20th International Symposium on Quality Electronic Design (ISQED)*, March 2019.

[23] Yang Zhang, Bilal Anwer, Vijay Gopalakrishnan, Bo Han, Joshua Reich, Aman Shaikh, and Zhi-Li Zhang. Parabox: Exploiting parallelism for virtual network functions in service chaining. In *Proceedings of the Symposium on SDN Research*, pages 143–149. ACM, 2017.

[24] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Francis Matus, Rong Pan, Navindra Yadav, George Varghese, et al. Conga: Distributed congestion-aware

load balancing for datacenters. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 503–514. ACM, 2014.

[25] Yang Zhang, Hesham Mekky, Zhi-Li Zhang, Fang Hao, Sarit Mukherjee, and TV Lakshman. Sampo: Online subflow association for multipath tcp with partial flow records. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9. IEEE, 2016.

[26] Yang Zhang, Bo Han, Zhi-Li Zhang, and Vijay Gopalakrishnan. Network-assisted raft consensus algorithm. In *Proceedings of the SIGCOMM Posters and Demos*, pages 94–96. ACM, 2017.

[27] Yang Zhang, Eman Ramadan, Hesham Mekky, and Zhi-Li Zhang. When raft meets sdn: How to elect a leader and reach consensus in an unruly network. In *Proceedings of the First Asia-Pacific Workshop on Networking*, pages 1–7. ACM, 2017.

[28] Xiongzi Ge. Improving data management and data movement efficiency in hybrid storage systems. 2017.

[29] Xiongzi Ge, Xuchao Xie, David HC Du, Pradeep Ganesan, and Dennis Hahn. Chewanalyzer: Workload-aware data management across differentiated storage pools. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 94–101. IEEE, 2018.

[30] Bingzhe Li, Manas Minglani, and David Lilja. Ps-code: A new code for improved degraded mode read and write performance of raid systems. In *Networking, Architecture and Storage (NAS), 2016 IEEE International Conference on*, pages 1–10. IEEE, 2016.

[31] Bingzhe Li, Meng Yang, Soheil Mohajer, Weikang Qian, and David Lilja. Tier-code: An xor-based raid-6 code with improved write and degraded-mode read performance. In *Networking, Architecture and Storage (NAS), 2018 IEEE International Conference on*. IEEE, 2018.

[32] Yi Liu, Xiongzi Ge, Xiaoxia Huang, and David HC Du. Molar: A cost-efficient, high-performance ssd-based hybrid storage cache. *The Computer Journal*, 58(9):2061–2078, 2015.

[33] Bingzhe Li and David Du. Tasecure: Temperature-aware secure deletion scheme for solid state drives. In *The 29th edition of the ACM Great Lakes Symposium on VLSI (GLSVLSI'19)*. ACM, 2018.

[34] Zhichao Cao, Hao Wen, Fenggang Wu, and David H.C. Du. ALACC: Accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 309–324, Oakland, CA, 2018. USENIX Association.

[35] Zhichao Cao, Hao Wen, Xiongzi Ge, Jingwei Ma, Jim Diehl, and David H. C. Du. Tddfs: A tier-aware data deduplication-based file system. *ACM Trans. Storage*, 15(1):4:1–4:26, February 2019.

[36] Zhichao Cao, Shiyong Liu, Fenggang Wu, Guohua Wang, Bingzhe Li, and David H.C. Du. Sliding look-back window assisted data chunk rewriting for improving deduplication restore performance. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 129–142, Boston, MA, 2019. USENIX Association.

[37] M. Minglani, J. Diehl, X. Cao, B. Li, D. Park, D. J. Lilja, and D. H. C. Du. Kinetic action: Performance analysis of integrated key-value storage devices vs. leveldb servers. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 501–510, Dec 2017.

[38] Fenggang Wu, Baoquan Zhang, zhichao Cao, Hao Wen, Bingzhe Li, Jim Diehl, Guohua Wang, and David H.C. Du. Data management design for interlaced magnetic recording. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, Boston, MA, 2018. USENIX Association.

[39] F. Wu, Z. Fan, M. Yang, B. Zhang, X. Ge, and D. H. C. Du. Performance evaluation of host aware shingled magnetic recording (ha-smr) drives. *IEEE Transactions on Computers*, 66(11):1932–1945, Nov 2017.

[40] Fenggang Wu, Bingzhe Li, zhichao Cao, Baoquan Zhang, Ming-Hong Yang, Hao Wen, and David H.C. Du. Zonealloy: Elastic data and space management for hybrid SMR drives. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, 2019. USENIX Association.

[41] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[42] Bingzhe Li, M Hassan Najafi, and David J Lilja. Using stochastic computing to reduce the hardware requirements for a restricted boltzmann machine classifier. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 36–41. ACM, 2016.

[43] Bingzhe Li, M Hassan Najafi, and David J Lilja. An fpga implementation of a restricted boltzmann machine classifier using stochastic bit streams. In *Application-specific Systems, Architectures and Processors (ASAP), 2015 IEEE 26th International Conference on*, pages 68–69. IEEE, 2015.

[44] Bingzhe Li, Yaobin Qin, Bo Yuan, and David J Lilja. Neural network classifiers using stochastic computing with a hardware-oriented approximate activation function. In *2017 IEEE 35th International Conference on Computer Design (ICCD)*, pages 97–104. IEEE, 2017.

[45] Bingzhe Li, M Hassan Najafi, Bo Yuan, and David J Lilja. Quantized neural networks with new stochastic multipliers. In *2018 19th International Symposium on Quality Electronic Design (ISQED)*, pages 376–382. IEEE, 2018.

[46] Yaobin Qin, Bingzhe Li, and David. J Lilja. Enhancing the ensemble of exemplar-svms for binary classification using concurrent selection and ensemble learning. In *Ubiquitous Computing, Electronics and Mobile Communication Conference (UEMCON), 2018 IEEE 9th Annual*. IEEE, 2018.

[47] Bingzhe Li, Jiaxi Hu, M.Hassan Najafi, Steven Koester, and David. J Lilja. Low cost hybrid spin-cmos compressor for stochastic neural networks. In *The 29th edition of the ACM Great Lakes Symposium on VLSI (GLSVLSI'19)*. ACM, 2018.

[48] Bingzhe Li, M Hassan Najafi, and David J Lilja. Low-cost stochastic hybrid multiplier for quantized neural networks. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 15(2):18, 2019.

[49] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.

[50] Pang-Ning Tan. *Introduction to data mining*. Pearson Education India, 2018.

[51] Yu Zheng. Trajectory data mining: an overview. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 6(3):29, 2015.

[52] Shashi Shekhar, Zhe Jiang, Reem Ali, Emre Eftelioglu, Xun Tang, Venkata Gunturi, and Xun Zhou. Spatiotemporal data mining: a computational perspective. *ISPRS International Journal of Geo-Information*, 4(4):2306–2338, 2015.

[53] Yiqun Xie, Emre Eftelioglu, Reem Ali, Xun Tang, Yan Li, Ruhi Doshi, and Shashi Shekhar. Transdisciplinary foundations of geospatial data science. *ISPRS International Journal of Geo-Information*, 6(12):395, 2017.

[54] Xun Tang, Emre Eftelioglu, Dev Oliver, and Shashi Shekhar. Significant linear hotspot discovery. *IEEE Transactions on Big Data*, 3(2):140–153, 2017.

[55] Emre Eftelioglu, Zhe Jiang, Xun Tang, and Shashi Shekhar. The nexus of food, energy, and water resources: Visions and challenges in spatial computing. In *Advances in geocomputation*, pages 5–20. Springer, 2017.

[56] Bingzhe Li, Farnaz Toussi, Clark Anderson, David J Lilja, and David HC Du. Tracerar: An i/o performance evaluation tool for replaying, analyzing, and regenerating traces. In *Networking, Architecture, and Storage (NAS), 2017 International Conference on*, pages 1–10. IEEE, 2017.

[57] Bingzhe Li, Hao Wen, Farnaz Toussi, Clark Anderson, Bernard A King-Smith, David J Lilja, and David HC Du. Netstorage: A synchronized trace-driven replayer for network-storage system evaluation. *Performance Evaluation*, 130:86–100, 2019.

[58] Hao Wen, David HC Du, Milan Shetti, Doug Voigt, and Shanshan Li. Guaranteed bang for the buck: Modeling vdi applications with guaranteed quality of service. In *Parallel Processing (ICPP), 2016 45th International Conference on*, pages 426–431. IEEE, 2016.

[59] Hao Wen, David HC Du, Milan Shetti, Doug Voigt, and Shanshan Li. Guaranteed bang for the buck: Modeling vdi applications to identify storage requirements. *IEEE Transactions on Cloud Computing*, 2018.

[60] Docker documentation. `https://docs.docker.com/`. Accessed: 2019-2-26.

[61] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[62] Hao Wen, Zhichao Cao, Yang Zhang, Xiang Cao, Ziqi Fan, Doug Voigt, and David Du. Joins: Meeting latency slo with integrated control for networked storage. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 194–200. IEEE, 2018.

[63] Microsoft onedrive. `https://onedrive.live.com/about/en-us`. Accessed: 2019-2-26.

[64] Apple icloud. `https://www.apple.com/icloud`. Accessed: 2019-2-26.

[65] Google drive. `https://www.google.com/drive`. Accessed: 2019-2-26.

[66] Vdi virtual desktop infrastructure with horizon. `https://www.vmware.com/products/horizon-view`.

[67] Desktop virtualisation. `https://www.microsoft.com/en-in/cloud-platform/desktop-virtualization`. Accessed: 2017-10-01.

[68] Citrix virtual desktop handbook 7.x. https://support.citrix.com/article/CTX221865, 2017. Accessed: 2017-9-29.

[69] Vmware horizon 6 storage considerations. https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/vmware-horizon-view-mirage-workspace-portal-app-volumes-storage.pdf. Accessed: 2018-10-17.

[70] VMware Infrastructure. Vdi server sizing and scaling. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.134.7129&rep=rep1&type=pdf. Accessed: 2018-10-17.

[71] Server and storage sizing guide for windows 7 desktops in a virtual desktop infrastructure. https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/technicalnote/view/server-storage-sizing-guide-windows-7-technical-note.pdf. Accessed: 2018-10-17.

[72] Sizing and best practices for deploying vmware view 5.1 on vmware vsphere 5.0 u1 with dell equallogic storage. https://downloads.dell.com/manuals/all-products/esuprt_solutions_int/esuprt_solutions_int_solutions_resources/s-solution-resources_white-papers71_en-us.pdf. Accessed: 2018-10-17.

[73] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

[74] Daniel Gmach, Jerry Rolia, Ludmila Cherkasova, and Alfons Kemper. Workload analysis and demand prediction of enterprise data center applications. In *Workload Characterization, 2007. IISWC 2007. IEEE 10th International Symposium on*, pages 171–180. IEEE, 2007.

[75] Hongliang Yu, Dongdong Zheng, Ben Y Zhao, and Weimin Zheng. Understanding user behavior in large-scale video-on-demand systems. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 333–344. ACM, 2006.

[76] Viewplannerusersguide3.6.0. https://my.vmware.com/en/group/vmware/details?downloadGroup=VIEW-PLAN-360&productId=320. Accessed: 2019-2-26.

[77] Iomark workloads. http://www.iomark.org/content/workloads. Accessed: 2019-2-26.

[78] Vmware virtual san design and sizing guide for horizon view virtual desktop infrastructures. https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/whitepaper/products/vsan/vmw-tmd-virt-san-dsn-szing-guid-horizon-view-white-paper.pdf. Accessed: 2019-2-26.

[79] Spc-1 and spc-1e benchmark results. http://spcresults.org/benchmarks/results/spc1-spc1e. Accessed: 2019-2-26.

[80] Spc benchmark 1/energy extension official specification. https://spcresults.org/sites/default/files/files/specifications/SPC-1_SPC-1E_v1.14.pdf. Accessed: 2019-2-26.

[81] runc. https://github.com/opencontainers/runc. Accessed: 2019-2-26.

[82] View storage accelerator in vmware view 5.1. https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/vmware-view-storage-accelerator-host-caching-content-based-read-cache-white-paper.pdf. Accessed: 2019-2-26.

[83] Infinio. http://www.infinio.com. Accessed: 2019-2-26.

[84] Optimize vdi with server-side storage acceleration. https://blogs.networld.co.jp/main/files/PernixData_Optimize_VDI_WP.pdf. Accessed: 2019-2-26.

[85] Sriram Sankar and Kushagra Vaid. Storage characterization for unstructured data in online services applications. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 148–157. IEEE, 2009.

[86] Christina Delimitrou, Sriram Sankar, Kushagra Vaid, and Christos Kozyrakis. Accurate modeling and generation of storage i/o for datacenter workloads. *Proc. of EXERT, CA*, 2011.

[87] Philip Carns, Kevin Harms, William Allcock, Charles Bacon, Samuel Lang, Robert Latham, and Robert Ross. Understanding and improving computational science storage access through continuous characterization. *ACM Transactions on Storage (TOS)*, 7(3):8, 2011.

[88] Christopher Muelder, Carmen Sigovan, Kwan-Liu Ma, Jason Cope, Sam Lang, Kamil Iskra, Pete Beckman, and Robert Ross. Visual analysis of i/o system behavior for high-end computing. In *Proceedings of the third international workshop on Large-scale system and application performance*, pages 19–26. ACM, 2011.

[89] Weiping He, David HC Du, and Sai B Narasimhamurthy. Pioneer: A solution to parallel i/o workload characterization and generation. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 111–120. IEEE, 2015.

[90] Vasily Tarasov, Dean Hildebrand, Geoff Kuenning, and Erez Zadok. Virtual machine workloads: the case for new benchmarks for nas. In *FAST*, pages 307–320, 2013.

[91] Ajay Gulati, Chethan Kumar, and Irfan Ahmad. Storage workload characterization and consolidation in virtualized environments. In *Workshop on Virtualization Performance: Analysis, Characterization, and Tools (VPACT)*, 2009.

[92] Asit K Mishra, Joseph L Hellerstein, Walfredo Cirne, and Chita R Das. Towards characterizing cloud backend workloads: insights from google compute clusters. *ACM SIGMETRICS Performance Evaluation Review*, 37(4):34–41, 2010.

[93] Ajay Gulati, Chethan Kumar, and Irfan Ahmad. Modeling workloads and devices for io load balancing in virtualized environments. *ACM SIGMETRICS Performance Evaluation Review*, 37(3):61–66, 2010.

[94] C. Le Thanh Man and M. Kayashima. Virtual machine placement algorithm for virtualized desktop infrastructure. In *2011 IEEE International Conference on Cloud Computing and Intelligence Systems*, pages 333–337, 2011.

[95] vsan. `https://www.vmware.com/products/vsan.html`. Accessed: 2019-2-26.

[96] Vmware vsphere. `https://docs.vmware.com/en/VMware-vSphere/index.html#com.vmware.vsphere.doc`. Accessed: 2019-2-26.

[97] Hp 3par storeserv storage concepts guide. `https://support.hpe.com/hpsc/doc/public/display?docId=c04204225`. Accessed: 2019-2-26.

[98] Joe Beda. Containers at scale. `https://speakerdeck.com/jbeda/containers-at-scale?slide=2`. Accessed: 2019-2-26.

[99] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. Ioflow: A software-defined storage architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 182–196, 2013.

[100] Raúl Gracia-Tinedo, Josep Sampé, Edgar Zamora, Marc Sánchez-Artigas, Pedro García-López, Yosef Moatti, and Eran Rom. Crystal: Software-defined storage for multi-tenant object stores. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, pages 243–256. USENIX Association, 2017.

[101] Ajay Gulati, Ganesha Shanmuganathan, Irfan Ahmad, Carl Waldspurger, and Mustafa Uysal. Pesto: online storage performance management in virtualized datacenters. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 19. ACM, 2011.

[102] Persistent volumes. `https://kubernetes.io/docs/concepts/storage/persistent-volumes/`. v1.13. Accessed: 2019-2-26.

[103] Scaling. https://kubernetes.io/docs/getting-started-guides/ubuntu/scaling/. Accessed: 2019-2-26.

[104] Building large clusters. https://kubernetes.io/docs/setup/cluster-large/#size-of-master-and-master-components. Accessed: 2019-2-26.

[105] Flexvolume. https://github.com/kubernetes/community/blob/master/contributors/devel/flexvolume.md. Accessed: 2019-2-26.

[106] Paul Menage, Paul Jackson, and Christoph Lameter. Cgroups. *Available on-line at: http://www.mjmwired.net/kernel/Documentation/cgroups.txt*, 2008.

[107] Nginx. https://www.nginx.com/. Accessed: 2019-2-26.

[108] Openstack swift documentation. https://docs.openstack.org/swift/latest/.

[109] Swiftstack benchmark suite (ssbench). https://github.com/swiftstack/ssbench. Accessed: 2019-2-26.

[110] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, pages 171–172. IEEE, 2015.

[111] EMC Dell. Improving copy-on-write performance in container storage drivers. https://www.snia.org/sites/default/files/SDC/2016/presentations/capacity_optimization/FrankZaho_Improving_COW_Performance_ContainerStorage_Drivers-Final-2.pdf. Accessed: 2019-2-26.

[112] Vasily Tarasov, Lukas Rupprecht, Dimitris Skourtis, Amit Warke, Dean Hildebrand, Mohamed Mohamed, Nagapramod Mandagere, Wenji Li, Raju Rangaswami, and Ming Zhao. In search of the ideal storage configuration for docker containers. In *Foundations and Applications of Self* Systems (FAS* W), 2017 IEEE 2nd International Workshops on*, pages 199–206. IEEE, 2017.

[113] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Slacker: Fast distribution with lazy docker containers. In *FAST*, volume 16, pages 181–195, 2016.

[114] Ali Anwar, Mohamed Mohamed, Vasily Tarasov, Michael Littley, Lukas Rupprecht, Yue Cheng, Nannan Zhao, Dimitrios Skourtis, Amit S Warke, Heiko Ludwig, et al. Improving docker registry design based on production workload analysis. In *16th USENIX Conference on File and Storage Technologies*, page 265, 2018.

[115] Gaurav Makin, Kody Kantor, Hao Wen, Zhichao Cao, and Vallari Mehta. Systems and methods for performing live migrations of software containers, December 25 2018. US Patent App. 10/162,559.

[116] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. ACM, 2015.

[117] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Queue*, 14(1):10, 2016.

[118] The history of kubernetes on a timeline. `https://blog.risingstack.com/the-history-of-kubernetes/`. Accessed: 2019-2-26.

[119] Víctor Medel, Omer Rana, José Ángel Bañares, and Unai Arronategui. Modelling performance & resource management in kubernetes. In *2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC)*, pages 257–262. IEEE, 2016.

[120] Víctor Medel, Omer Rana, José Ángel Bañares, and Unai Arronategui. Adaptive application scheduling under interference in kubernetes. In *Utility and Cloud Computing (UCC), 2016 IEEE/ACM 9th International Conference on*, pages 426–427. IEEE, 2016.

[121] Cong Xu, Karthick Rajamani, and Wesley Felter. Nbwguard: Realizing network qos for kubernetes. In *Proceedings of the 19th International Middleware Conference Industry*, pages 32–38. ACM, 2018.

[122] Pei-Hsuan Tsai, Hua-Jun Hong, An-Chieh Cheng, and Cheng-Hsin Hsu. Distributed analytics in fog computing platforms using tensorflow and kubernetes. In *Network Operations and Management Symposium (APNOMS), 2017 19th Asia-Pacific*, pages 145–150. IEEE, 2017.

[123] Rex-ray. `https://rexray.readthedocs.io/en/stable/`. Accessed: 2019-2-26.

[124] Trident. `https://netapp-trident.readthedocs.io/en/stable-v18.10/`. Accessed: 2019-2-26.

[125] Netapp/trident github. `https://github.com/NetApp/trident`. Accessed: 2019-2-26.

[126] Vmware storage drs. `https://docs.vmware.com/en/VMware-vSphere/6.0/com.vmware.vsphere.resmgmt.doc/GUID-827DBD6D-08B7-4411-9214-9E126671457F.html`. Accessed: 2019-3-28.

[127] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy Katz, and Ion Stoica. Cake: Enabling high-level slos on shared storage systems. In *ACM Symposium on Cloud Computing*, pages 1–14, 2012.

[128] Timothy Zhu, Alexey Tumanov, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. Prioritymeister: Tail latency qos for shared networked storage. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14, 2014.

[129] Ioan Stefanovici, Bianca Schroeder, Greg O'Shea, and Eno Thereska. sroute: Treating the storage stack like a network. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 197–212, 2016.

[130] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to sdn. *Queue*, 11(12):20, 2013.

[131] Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. In *ACM SIGCOMM Computer Communication Review*, pages 1–12. ACM, 2007.

[132] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. Nox: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, 2008.

[133] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[134] Sean McDaniel, Stephen Herbein, and Michela Taufer. A two-tiered approach to i/o quality of service in docker containers. In *2015 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 490–491. IEEE, 2015.

[135] Microsoft Corporation. Server message block (smb) protocol. `https://winprotocoldoc.blob.core.windows.net/productionwindowsarchives/MS-SMB/[MS-SMB].pdf`. Release: June 1, 2017.

[136] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. End-to-end performance isolation through virtual datacenters. In *OSDI*, pages 233–248, 2014.

[137] Zhe Wu, Curtis Yu, and Harsha V. Madhyastha. Costlo: Cost-effective redundancy for lower latency variance on cloud storage services. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, pages 543–557, 2015.

[138] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking (ToN)*, 1(4):397–413, 1993.

[139] Sally Floyd. Tcp and explicit congestion notification. *ACM SIGCOMM Computer Communication Review*, 24(5):8–23, 1994.

[140] Yingping Lu, David HC Du, Chuanyi Liu, and Xianbo Zhang. Qos scheduling for networked storage system. In *Distributed Computing Systems, 2008. ICDCS'08. The 28th International Conference on*, pages 605–612. IEEE, 2008.

[141] Walid G Aref, Khaled El-Bassyouni, Ibrahim Kamel, and Mohamed F Mokbel. Scalable qos-aware disk-scheduling. In *Database Engineering and Applications Symposium, 2002. Proceedings. International*, pages 256–265. IEEE, 2002.

[142] Abhay K Parekh and Robert G Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM transactions on networking*, 1(3):344–357, 1993.

[143] Ye Wang, Hao Wang, Ajay Mahimkar, Richard Alimi, Yin Zhang, Lili Qiu, and Yang Richard Yang. R3: resilient routing reconfiguration. In *ACM SIGCOMM Computer Communication Review*, volume 40, pages 291–302. ACM, 2010.

[144] Sanghwan Lee, Yinzhe Yu, Srihari Nelakuditi, Zhi-Li Zhang, and Chen-Nee Chuah. Proactive vs reactive approaches to failure resilient routing. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 1. IEEE, 2004.

[145] Matthew P Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert NM Watson, Andrew W Moore, Steven Hand, and Jon Crowcroft. Queues don't matter when you can jump them! In *NSDI*, pages 1–14, 2015.

[146] Andrew Tanenbaum. *Computer Networks*, chapter 1, pages 161–164. Prentice Hall Professional Technical Reference, 2011.

[147] Seagate. Scsi commands reference manual. `https://www.seagate.com/files/staticfiles/support/docs/manual/Interface%20manuals/100293068j.pdf`. Release: October, 2016.

[148] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 174–187. ACM, 2001.

[149] Yu Hua, Xue Liu, and Dan Feng. Smart in-network deduplication for storage-aware sdn. *ACM SIGCOMM Computer Communication Review*, 43(4):509–510, 2013.

[150] Xiongzi Ge, Yi Liu, Chengtao Lu, Jim Diehl, David HC Du, Liang Zhang, and Jian Chen. Vnre: Flexible and efficient acceleration for network redundancy elimination. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 83–92. IEEE, 2016.

[151] Xiongzi Ge, Yi Liu, David HC Du, Liang Zhang, Hongguang Guan, Jian Chen, Yuping Zhao, and Xinyu Hu. Openanfv: Accelerating network function virtualization with a consolidated framework in openstack. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 353–354. ACM, 2014.

[152] Kiran Srinivasan, Timothy Bisson, Garth R Goodson, and Kaladhar Voruganti. idedup: latency-aware, inline data deduplication for primary storage. In *FAST*, volume 12, pages 1–14, 2012.

[153] Open vswitch. `http://openvswitch.org`. Accessed: 2019-2-26.

[154] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)*, 4(3):10, 2008.

[155] Openstack block storage (cinder) documentation. `https://docs.openstack.org/cinder/latest/`.